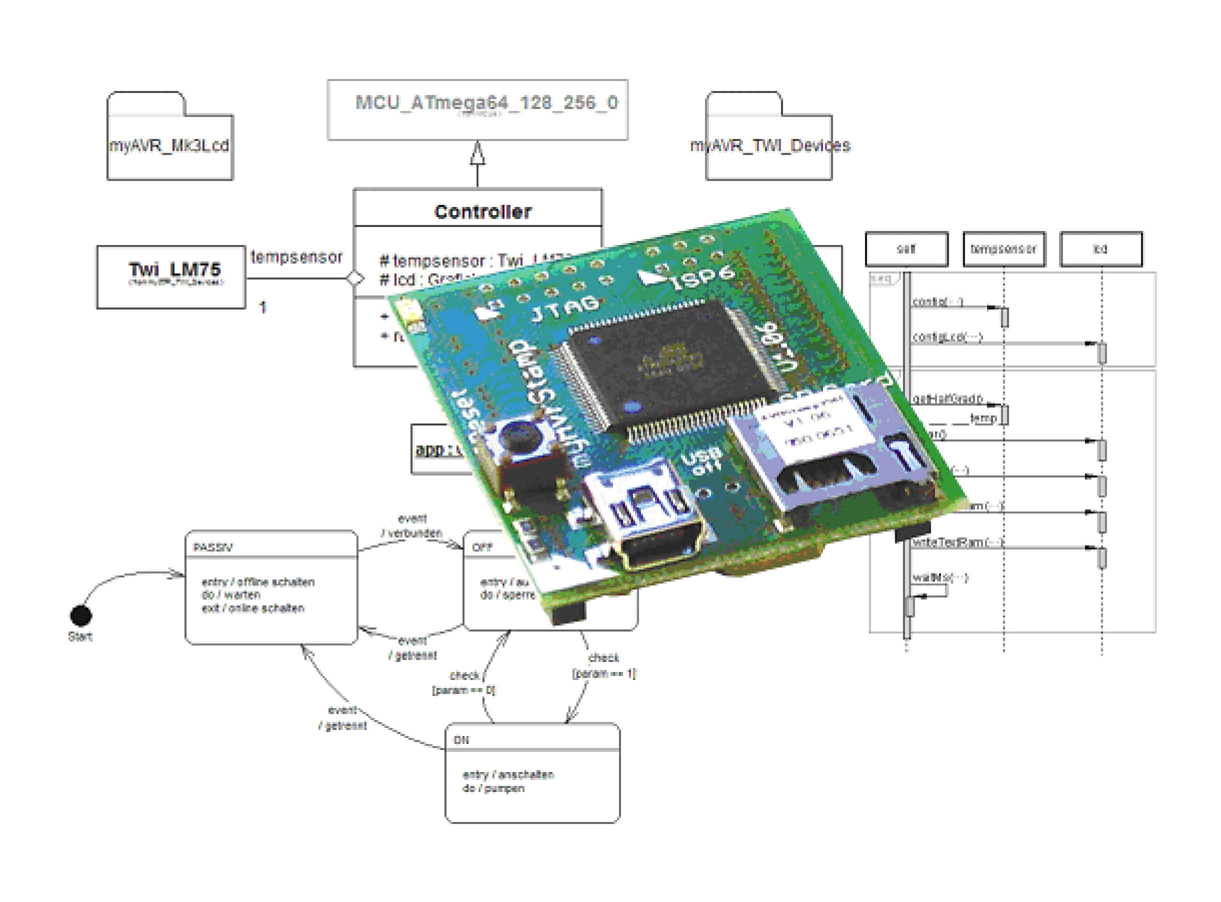
Sven Löbmann
Dipl. Ing. Toralf Riedel
Dipl. Ing. Päd. Alexander Huwaldt

Software Engineering für Embedded Systems

Ein myAVR Lehrbuch für die praxisorientierte Einführung
Software Engineering mit AVR-Mikrocontrollern



Leseprobe

1. Auflage: Dezember 2010

www.laser-co.de
www.myavr.de
info@myavr.de
Tel: ++49 (0) 3585 470 222
Fax: ++49 (0) 3585 470 233

Inhalt

Vorwort

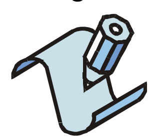
Dieses Buch wendet sich an Leser, die bereits über Kenntnisse in der Programmierung von AVR Mikrocontrollern verfügen. Die Autoren haben aus ihrer langjährigen Projekterfahrung mit kleinen, mittleren und großen Projekten, in diesem Lehrbuch wesentliche Aspekte der methodischen und systematischen Anwendung von Basiskonzepten, Methoden, Techniken und Werkzeugen der ingenieurmäßigen Herstellung von Software, also des Software Engineerings zusammengefasst. Besonderes Augenmerk gilt hier einer Sparte der Systementwicklung welche zunehmend an Bedeutung gewinnt und gegenüber der Programmierung von PC-Anwendungen oder dem Erstellen von Internetlösungen eine Reihe spezifischer Merkmale aufweist. Der Inhalt bezieht sich speziell auf Entwurf und Realisierung von Mikrocontrollerlösungen und orientiert sich damit auf die Bedürfnisse der Entwicklung von eingebetteten Systemen. Somit werden wichtige Aspekte der Softwareseite des Embedded Systems Engineering erörtert und angewendet. In die Gestaltung dieses Lehrbuches flossen ebenfalls die Anforderungen und Erfahrungen aus der Lehrtätigkeit der Autoren an Berufsakademien, Fachhochschulen und bei Erwachsenenqualifizierungen ein. Gerade hier liegt das Spannungsfeld. Die Ausbildung von Informatikern über Elektrotechniker bis zu Mechatronikern gestaltet sich mehr und mehr zur Herausforderung. Dem Informatiker wird viel Softwaretechnik in einer zunehmend von der Hardware abstrahierten Welt und faktisch keine Elektrotechnik vermittelt. Dem Elektrotechniker wird naturgemäß viel Elektrotechnik mit auf den Weg gegeben, aber für die Softwareentwicklung, obwohl oft hardwarenah vermittelt, fehlt die Zeit zur notwendigen Vertiefung. Mechatroniker klagen sowieso über die ungeheuere Stofffülle zwischen Mechanik, Hydraulik, Pneumatik, Elektrotechnik und Informatik.

„Grau, teurer Freund, ist alle Theorie und grün des Lebens goldner Baum.“
Faust der Tragödie erster Teil, Johann Wolfgang von Goethe

Oft wird Software Engineering und die in diesem Diskurs angebotenen Techniken und Werkzeuge als nett, aber in der Gier endlich seine Ideen in Quellcode zu wandeln, als hinderlich angesehen. Hier vertreten die Autoren den Standpunkt, dass eine Technik nur dann von praktischem Nutzen ist, wenn die Arbeitsergebnisse des einen Schrittes in geeigneter Form im nächsten Arbeitsschritt möglichst direkt weiterverwendet werden können und von Werkzeugen unterstützt werden. Dieser Sachverhalt wird in theoretischen Abhandlungen nicht erlebbar. Deshalb basieren dieses Lehrbuch und der Lernerfolg auf der praktischen Anwendung des Vorgestellten und konsequenten Werkzeugnutzung.

„Wir lernen, was wir tun“
John Dewey

Beachten Sie, dass die Aufgaben im Lehrbuch in der Regel praktische Übungen mit dem myAVR Board MK3 und dem Entwicklungswerkzeug SiSy sind. Auf praktische Übungen werden Sie mit dem folgenden Symbol hingewiesen:



Weitere Informationen und Beispiele finden Sie unter
www.myAVR.de.

1 Einführung

„Was immer du tun kannst oder träumst es zu können, fang damit an.“
Johann Wolfgang von Goethe

Als ein eingebettetes System (Embedded System) werden Digitalrechner verstanden, welche von dem System das sie überwachen, steuern und regeln sollen unmittelbar umgeben werden. Oft ist das Vorhandensein eines Computers (Mikrocontrollers) in derartigen Systemen auf den ersten Blick nicht ersichtlich.

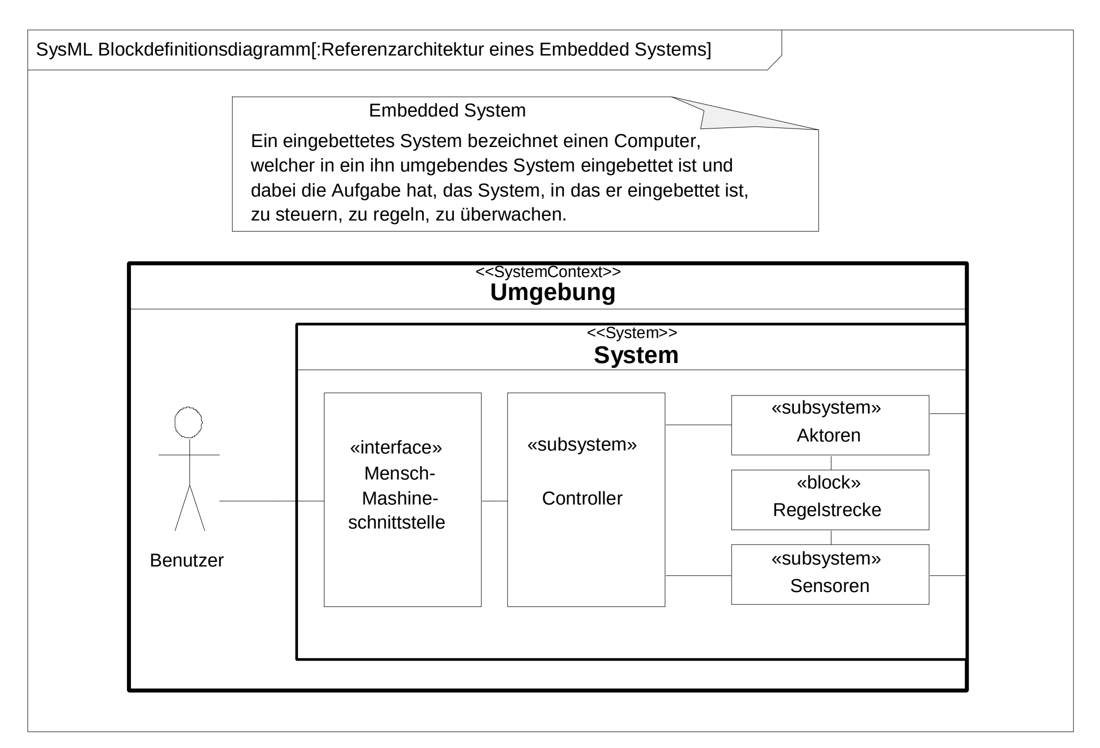


Abbildung: Referenzarchitektur eingebetteter Systeme

Die rein zahlenmäßige Menge der eingebetteten Systeme und rasenden Verbreitung derartiger Lösungen führen jedoch zu zunehmendem Interesse an diesen. Vom professionellen Anwendungsbereich bis zu Hobby und Kunst erfreuen sich Mikrocontroller immer größerer Beliebtheit. Der Buchmarkt wird überschwemmt von Werken, die den Einstieg in die Programmierung von Mikrocontrollern leichter machen sollen. Gleichzeitig nehmen Wissen, Leistungsfähigkeit der Systeme, Systemkomplexität und Projektgröße zu. Damit geht einher, dass die Quelltextgröße der dafür nötigen Software sich zunehmend der Beherrschbarkeit des Entwicklers entzieht. Moderne Werkzeuge bieten dem Entwickler neue Techniken an, die vor allem als grafische Programmiersprachen verstanden werden müssen. Die UML (Unified Modeling Language) und die SysML (Systems Modeling Language) markieren momentan und gewiss auch in den nächsten Jahren den Stand der Kunst. Folgen Sie uns auf dem Weg vom Basteln zum Konstruieren.

Die Arbeitsweise mit diesem Lehrbuch kann dem individuellen Wissensstand angepasst sein. Die Kapitel sind der generellen Reihenfolge nach aufeinander aufbauend, aber im Detail auch einzeln nachvollziehbar. Das gesamte Lehrbuch bezieht sich in den praktischen Übungen auf das myAVR Board MK3 als Referenzhardware und SiSy AVR /SiSy AVR++ als Modellierungswerkzeug.

Wenden wir uns zuerst einem kleinen geschichtlichen Abriss des Software-Engineering zu.

.....

1.4 Problemstellung

„Nichts ist schwieriger als das Vereinfachen.
Nichts ist einfacher als das Komplizieren.“
Georges Elgozy

Faktisch alle Aspekte des Software Engineering zielen auf die Beherrschbarkeit von Komplexität. Das ist natürlich keine Eigenheit der Softwaretechnik. Komplexität gibt es in allen Bereichen der menschlichen Aneignung der Welt. Das Besondere an der Informatik ist ihr zartes Alter von vielleicht 70 oder 80 Jahren. Damit hinkt sie wie ein kleines Kind vielen anderen Ingenieurdisziplinen um Jahrhunderte an Erfahrung hinterher.

Versuchen wir uns für das Problem der Komplexität zu sensibilisieren. Dazu führen wir ein kleines Experiment aus. Betrachten Sie die folgende Darstellung. Sie sollen die Menge der vorgelegten Streichhölzer ermitteln. Nehmen Sie sich nicht mehr als 3 Sekunden Zeit. In der Softwareergonomie gelten Reaktionszeiten von mehr als zwei Sekunden übrigens schon als eher unakzeptabel. Es geht also um das Wahrnehmen und das Verstehen.

Wie viele Streichhölzer sind das?

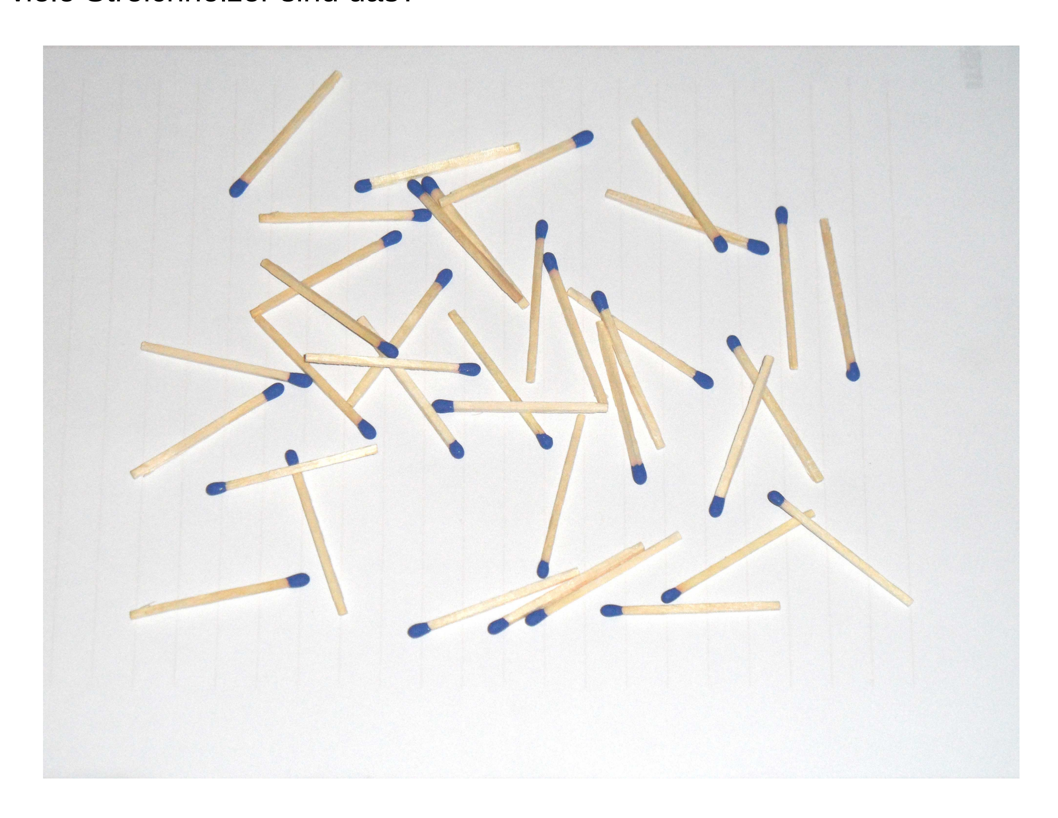


Abbildung: Streichholzexperiment 1. Versuch

In drei Sekunden werden Sie nicht mehr als eine Schätzung der Streichholzanzahl abgeben können. Wiederholen wir das Experiment mit einer neuen Darstellung.
Nehmen Sie sich wiederum nicht mehr als drei Sekunden Zeit die Anzahl der Hölzchen zu ermitteln.

Bitte jetzt umblättern!

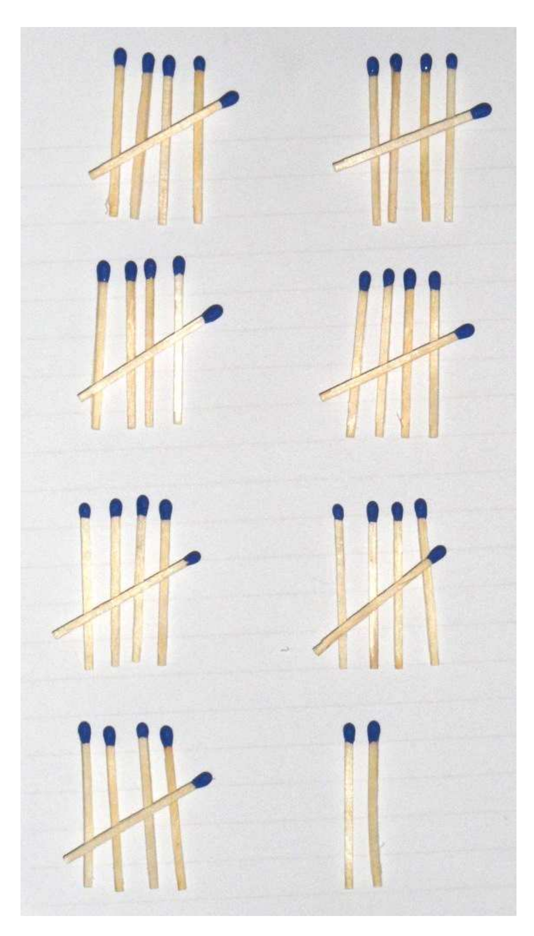


Abbildung: Streichholzexperiment 2. Versuch

Sie werden jetzt die exakte Anzahl der Streichhölzer in wesentlich weniger als drei Sekunden erfasst haben. Es hat vielleicht eine oder doch sogar noch fast zwei Sekunden gedauert die Anzahl zu erfassen. Trotzdem, Sie haben diesmal die gestellt Aufgabe in der geforderten Zeit exakt erfüllt. Irgendetwas war diesmal anders. Die Anzahl der Streichhölzer in beiden Versuchen war exakt die Selbe. Aber erst im zweiten Versuch waren Sie in der Lage, die Anzahl wirklich mit einem oder auch zwei Blicken zu erfassen. Woran lag das? Zum einen sorgte die Art der Anordnung, der Fachmann spricht hier von Strukturierung, dafür. Der zweite Aspekt ist der, dass Ihnen ein bekanntes Muster ja fast eine Art Symbol angeboten wurde.

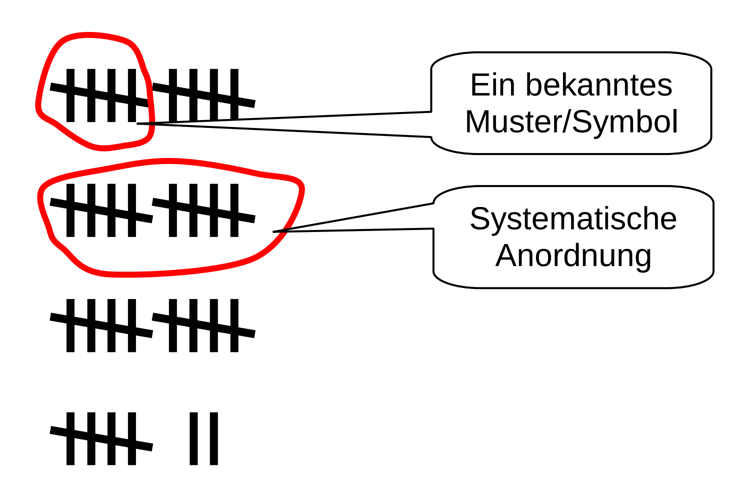


Abbildung: Verstehen durch Strukturierung

Des Weiteren wurde in der Art der Anordnung, durch denjenigen der die zweite Darstellung legt, eine Systematik hineingedacht die dem Betrachter nicht einmal erklärt werden musste, sondern die offensichtlich selbsterklärend wirkt (5er Symbol, 10er Gruppen, von links nach rechts, von oben nach unten). Hier scheint Unordnung gegen Ordnung zu stehen. Das ist zum einen durchaus richtig, zum anderen stellen wir uns einfach eine noch größere Anzahl Streichhölzer in der angebotenen Ordnung vor. Wir wären wieder nicht in der Lage, trotz Ordnung die Anzahl in drei Sekunden zu erfassen und der Platz für die bildliche Darstellung wäre recht üppig. Auch dieses Problem wurde schon lange gelöst. Man löst sich von dem Gedanken die Hölzer müssten als reale Gegenstände präsent sein. um deren Anzahl zu vermitteln. Der Weg führt zu abstrakten Symbolen die vereinbart, festgelegt und vor allem gelernt werden, um schnell und effizient Sachverhalte zu vermitteln. Sie erfassen und verstehen die Zahl 2011 in Bruchteilen von Sekunden. Dazu sind Sie in der Lage, weil Sie die Symbole und die Regeln ihrer Anwendung mühselig erlernt haben. Stellen Sie sich einfach die Kommunikation über Anzahl und Mengen, Längen, Zeiten usw. auf Streichholzniveau vor.

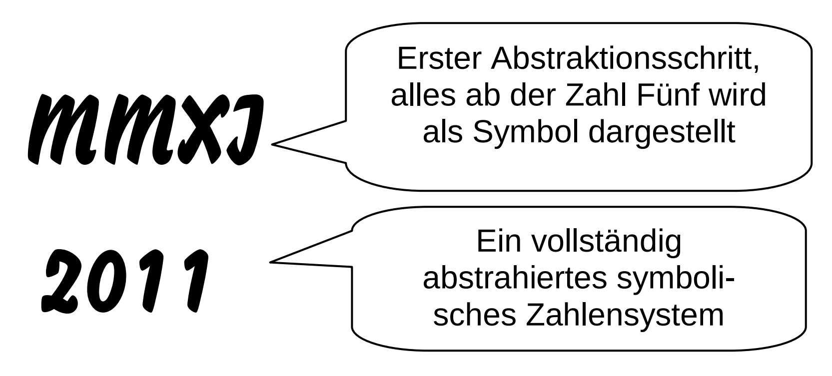


Abbildung: Verstehen durch zunehmende Abstraktion

Folgendes sollte sich jetzt offenbart haben:
- Ordnung, Strukturierung hilft Komplexität zu beherrschen
- Bekannte Muster, sich wiederholende Systematik hilft
- abstrakte Symbole sind entscheidende Elemente des Verstehens und der Kommunikation

Dringt man tiefer in die Problematik ein, findet sich sogar eine sozusagen goldene Strukturierungsregel. Wie der goldene Schnitt in der Malerei und Architektur gibt es bei der Strukturierung eine einfache Regel, wie Symbole anzuordnen oder wann noch abstraktere Symbole genutzt werden sollten. Der Schlüssel zum Verständnis findet sich in der Struktur unserer Wahrnehmung. Diese verläuft über das Ultrakurzzeitgedächtnis (auch sensorisches Gedächtnis). Hier werden viele 1000 Informationen in Bruchteil von Sekunden verarbeitet. Das geschieht unbewusst. Relevante Informationen werden an das Kurzzeitgedächtnis zur bewussten Verarbeitung weitergegeben. Hier können 3 bis 7 Informationen für Sekunden oder gar Minuten gehalten werden. Alles was wir von dort in unser Langzeitgedächtnis bringen, können wir uns für unbegrenzte Zeit merken. Der Flaschenhals ist das Kurzzeitgedächtnis mit seiner Kapazität von:

5 ± 2 Informationen

Abbildung: goldne Strukturierungsregel

Die 5 plus minus 2 Regel ist eine Faustformel, die dabei hilft Inhalte so zu strukturieren, dass diese schnell wahrgenommen und verstanden werden können. Diese Regel basiert auf der berühmten magischen 7 von George Armitage Miller. Seine Untersuchungen markierten die 7 als eine Obergrenze für unser Kurzzeitgedächtnis. So sollten in einem visuellen Zusammenhang etwa fünf bis maximal sieben jedoch nicht weniger als drei wesentliche Informationsblöcke dem Betrachter angeboten werden.

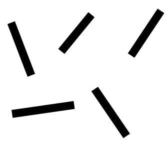


Abbildung: sofort überschaubare Struktur

Trotz Unordnung überblicken wir den Sachverhalt, denn der angebotene Inhalt passt in das Wahrnehmungsraster 5±2.

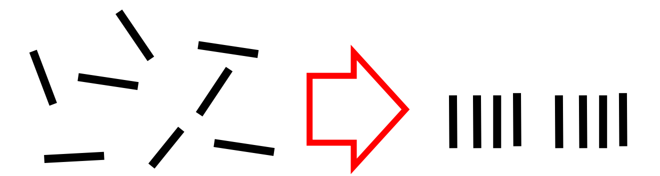


Abbildung: Strukturierung nach 5±2 Regel

Wird das Wahrnehmungsraster 5±2 überschritten, fällt uns das schnelle Erfassen schwer. Bringen wir den Inhalt in eine Struktur die wiederum dem 5±2 Prinzip folgt, nehmen wir es wieder schnell wahr. Oben wurden zwei Blöcke mit je vier Elementen angeordnet. Werden es mehr als sieben Blöcke wird es wieder schwierig.

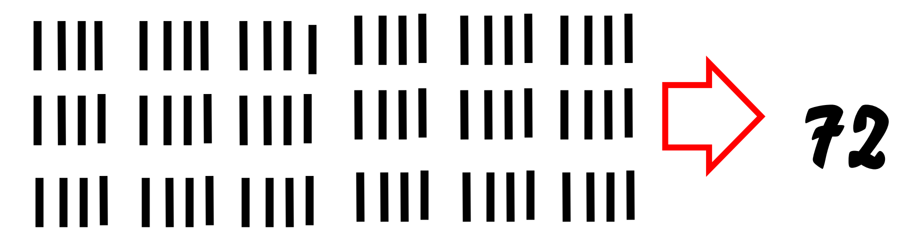


Abbildung: Reduktion der Anzahl an Elementen durch Abstraktion

Trotz der Strukturierung in Blöcken, zu je vier Elementen und drei Zeilen, überschreitet die Anzahl der Spalten das Wahrnehmungsraster 5±2 und zwingt uns zum mühseligen, langsamen und auch fehleranfälligen Zählen. Durch die Abstraktion der Elemente in Zahlensymbole wird der zu vermittelnde Inhalt wieder schnell erfassbar. Auf der Ebene der jetzt eingenommen Abstraktionsstufe gilt aber wiederum das 5±2 Prinzip.

72567809 72.567.809

Abbildung: 5±2 Regel auf der abstrakten Ebene

Der Tausenderpunkt strukturiert die Zahlenkolonne wieder nach dem 5±2 Prinzip und macht diese schnell erfassbar. Dieses Prinzip sollten sie verinnerlichen.

2 Entwicklungswerkzeug und Zielsystem

2.1 Die Entwicklungsumgebung

Die Bearbeitung der Übungen und Aufgaben bezieht sich auf die Verwendung des Modellierungswerkzeuges und Entwicklungsumgebung SiSy AVR / SiSy AVR++. Sollten Sie SiSy AVR / SiSy AVR++ bereits installiert haben, können Sie dieses Kapitel überspringen. Eine detaillierte Installationsbeschreibung für SiSy und die nötigen Treiber finden Sie unter www.myAVR.de in der Rubrik Download.

Voraussetzungen

Für die Installation benötigen Sie einen Freischaltcode (Lizenzangaben). Falls Sie diese Angaben nicht mit der Software erhalten haben, können Sie diese online abrufen von

www.myAVR.de → Online-Shop→ Kontakt/Service

oder fordern Sie diese beim Hersteller an:

Tel: 03585-470222
Fax: 03585-470233
e-Mail: hotline@myAVR.de.

Außerdem sollten Sie prüfen, ob die Systemvoraussetzungen für die Installation und die Arbeit mit SiSy AVR / SiSy AVR++ gewährleistet sind.

- PC-Arbeitsplatz oder Notebook mit USB-Anschluss
- Windows XP, Widows Vista oder Windows 7
- Mindestens 350 MB freier Speicherplatz auf der Festplatte
- Mindestens 1 GB RAM
- Microsoft Internet-Explorer ab Version 6.0
- Maus oder ähnliches Zeigegerät
- USB Kabel

Setup von der SiSy-CD

Legen Sie die CD „SiSy“ in Ihr CD-ROM-Laufwerk ein. Falls die CD nicht automatisch startet, wählen Sie bitte im Explorer das CD-ROM-Laufwerk und starten die *setup.exe* aus der Wurzel des Laufwerks.
Auf dem Startbildschirm stehen Schaltflächen zur Verfügung zum Installieren der Software und zum Öffnen von Begleitdokumenten.
Für die Installation der Software betätigen Sie die entsprechende Schaltfläche. In Abhängigkeit Ihrer Rechnerkonfiguration kann der Start des Setup-Programms einige Sekunden dauern. Das gestartete Setup-Programm wird Sie durch die weitere Installation führen.

Beginn der Installation

Betätigen Sie im Setup-Programm die Schaltfläche „Weiter“. Sie erhalten die Lizenzbestimmungen. Bitte lesen Sie diese sorgfältig durch. Wenn Sie sich mit diesen Bestimmungen einverstanden erklären, bestätigen Sie die Lizenzbestimmungen mit der Schaltfläche „Annehmen“.
Sie werden im folgenden Dialog dazu aufgefordert, Ihre Lizenzangaben einzugeben.

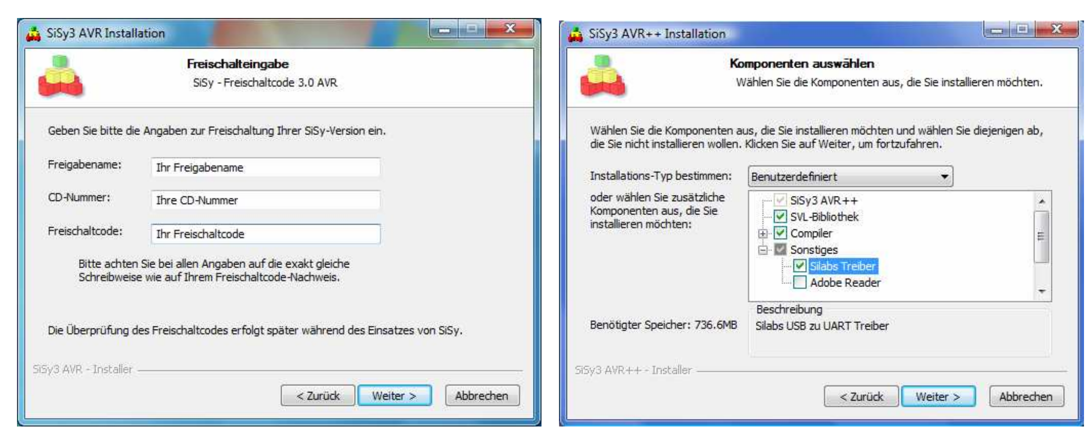


Abbildung: Eingabe der Lizenzdaten und Wahl der Treiberoption

Danach erscheint die Dialogbox „Komponenten auswählen", welche Sie mit „Weiter" bestätigen. Falls noch keine Treiber für die myAVR Hardware installiert sind wählen Sie bitte die Option „Sonstiges / Silabs Treiber". Im darauf folgenden Fenster können Sie festlegen, unter welchem Pfad SiSy AVR installiert werden soll. Wenn ein anderer Pfad (bzw. ein anderes Laufwerk) gewünscht wird, ist die Schaltfläche „Durchsuchen" zu betätigen. Eine Dialogbox erscheint, in der Sie Laufwerk und Verzeichnis auswählen können.



Abbildung: Auswahl des Installationsverzeichnisses

Bestimmen Sie danach den Startmenü-Ordner, in dem die Verknüpfungen von SiSy eingefügt werden. Sie können den Zielordner ändern. Sie können dies durch auswählen von „Keine Verknüpfungen erstellen" unterbinden.

Beginnen Sie nun die Installation durch betätigen der Schaltfläche „Installieren". Die Installation wird nach auswählen von „Fertig stellen" abgeschlossen.

Hinweis:
In SiSy sind 2 Dateien enthalten, die Makros beinhalten („handbuch.doc", „multi.doc"). Von einigen Virenscannern werden diese Makros als „Virus" erkannt und entsprechend behandelt. In den Heuristik-Einstellungen des Virenscanners kann diese Behandlung unterdrückt werden.

Sie können nun SiSy AVR starten. Es erscheint auf Ihren Bildschirm der Dialog „Willkommen in SiSy". Folgen Sie dann den Hinweisen des Assistenten, indem Sie „Assistent öffnen auswählen".

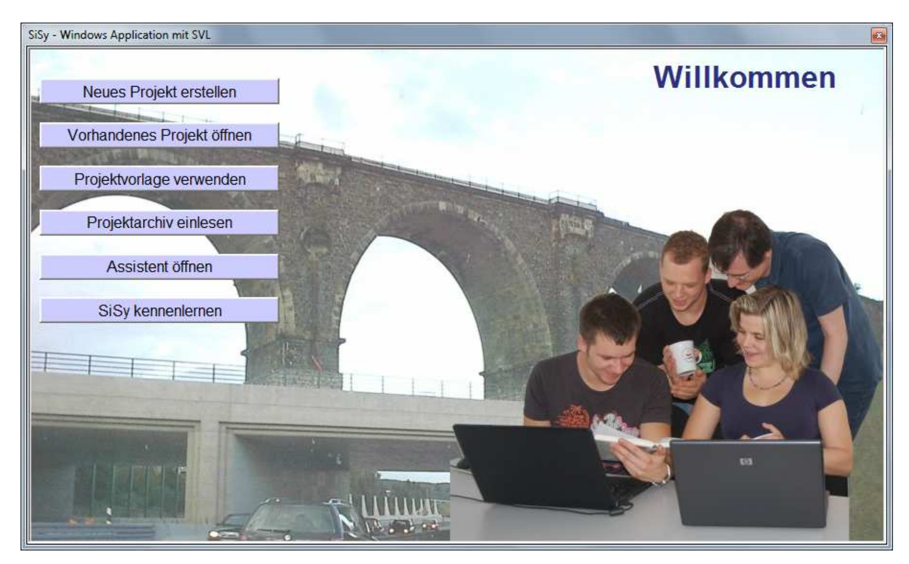


Abbildung: Startbildschirm von SiSy

Installation der Hardwaretreiber

Die myAVR Produkte mit USB Anschluss verfügen über einen CP2102 USB Controller der Firma Silicon Labs. Dabei handelt es sich um eine USB UART Bridge, die einen virtuellen COM-Port im System zur Verfügung stellt. Dieser kann wie ein normaler, physischer COM-Port benutzt werden. Die aktuellsten Treiber für die USB-Hardware finden Sie auf www.silabs.com. Sollten Sie die Treiber separat installieren gehen Sie bitte wie folgt vor.

Entpacken Sie die heruntergeladene Datei in ein temporäres Verzeichnis auf Ihrer Festplatte. Für eine reibungslose Installation schließen Sie das Board noch nicht an und starten das Treiberinstallationsprogramm.

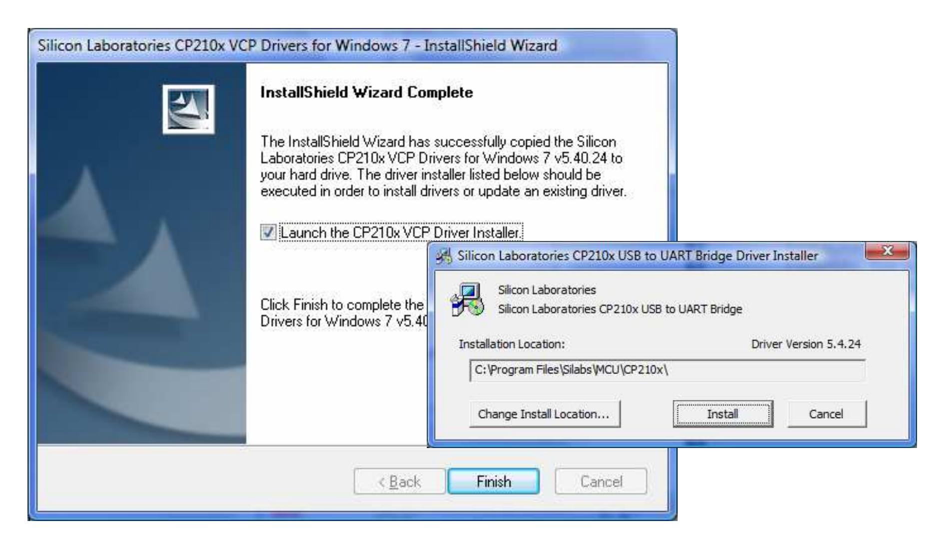


Abbildung: Start der Treiberinstallation

Der Treiber wird vorinstalliert und beim nächsten anschließen des myAVR Boards automatisch installiert. Sollten ältere Treiber vorhanden sein deinstallieren Sie diese bitte vorher.

2.2 Die Referenzhardware

Alle Ausführungen, Übungen und Aufgabenstellungen beziehen sich auf das myAVR Board MK3 als Zielsystem. Dieses ist im myAVR Aufsteigerset bereits enthalten oder kann einzeln erworben werden. Für das erfolgreiche Studium dieses Lehrbuches ist das physische Vorhandensein der beschriebenen Referenzhardware nicht zwingend notwendig aber empfohlen. Die beschriebenen Komponenten erhalten Sie unter www.myAVR.de.

Das myAVR Board MK3 ist ein leistungsfähiges Entwicklungsboard für Atmel Mikrocontroller der oberen Leistungsklasse. Es hat zahlreiche typische Hardware-Komponenten und Anschlussmöglichkeiten für die Entwicklung und das Testen von eingebetteten Systemen. Für das schnelle und unkomplizierte Arbeiten verfügt es über eine "quick connect option", d. h. alle Geräte können per Jumper sofort zugeschaltet, aber bei Bedarf auch frei verdrahtet werden. Das Board verfügt über vier Erweiterungsports für myAVR Add-Ons sowie eine Anschlussoption für ein myAVR Board MK1 LPT / MK2 USB oder einen mySmartControl MK2. Die Spannungsversorgung erfolgt im Normalfall über den USB-Anschluss kann aber bei Bedarf auch über ein externes Netzteil erfolgen. Das MK3 Board ist kompatibel zu allen myAVR Produkten. Die Programmierung erfolgt über den integrierten High-Speed-USB-Programmer mySmartUSB MK3.

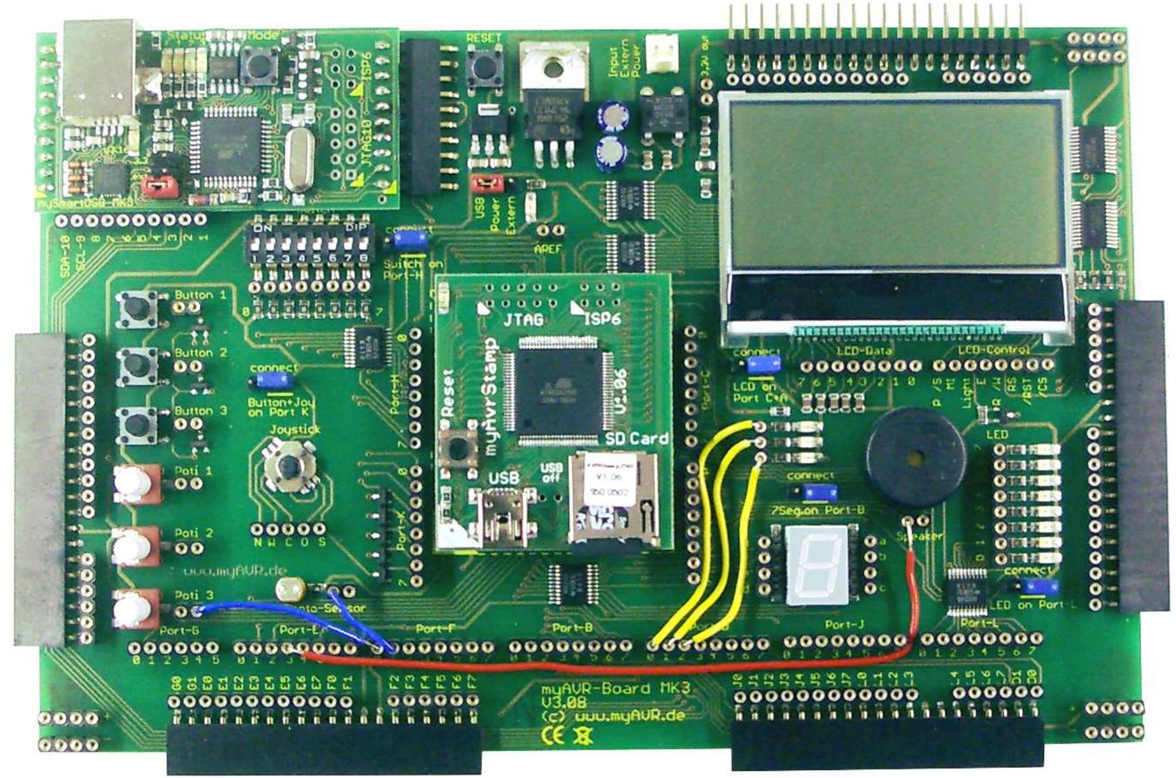


Abbildung: komplett bestücktes myAVR Board MK3

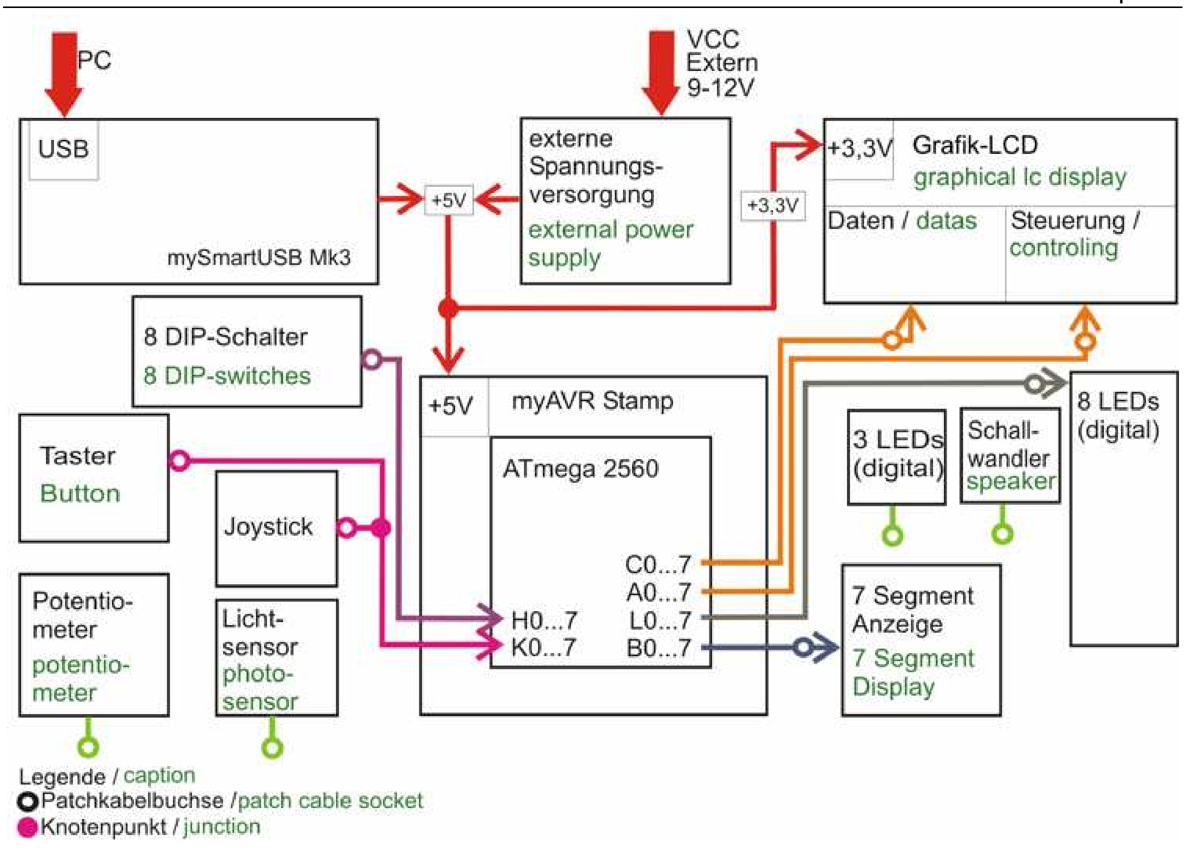


Abbildung: Blockbild des myAVR Board MK3

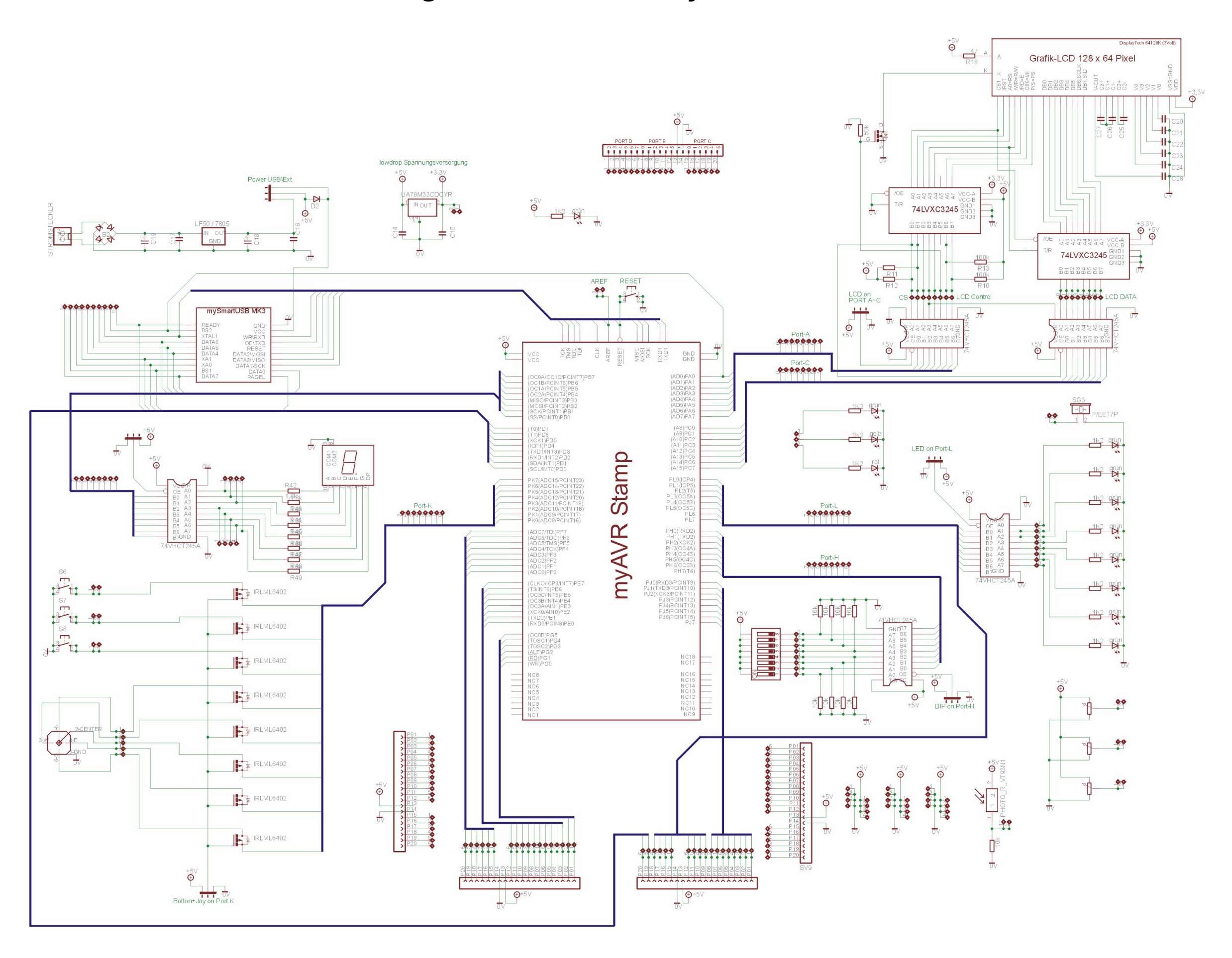


Abbildung: Schaltplan myAVR Board MK3

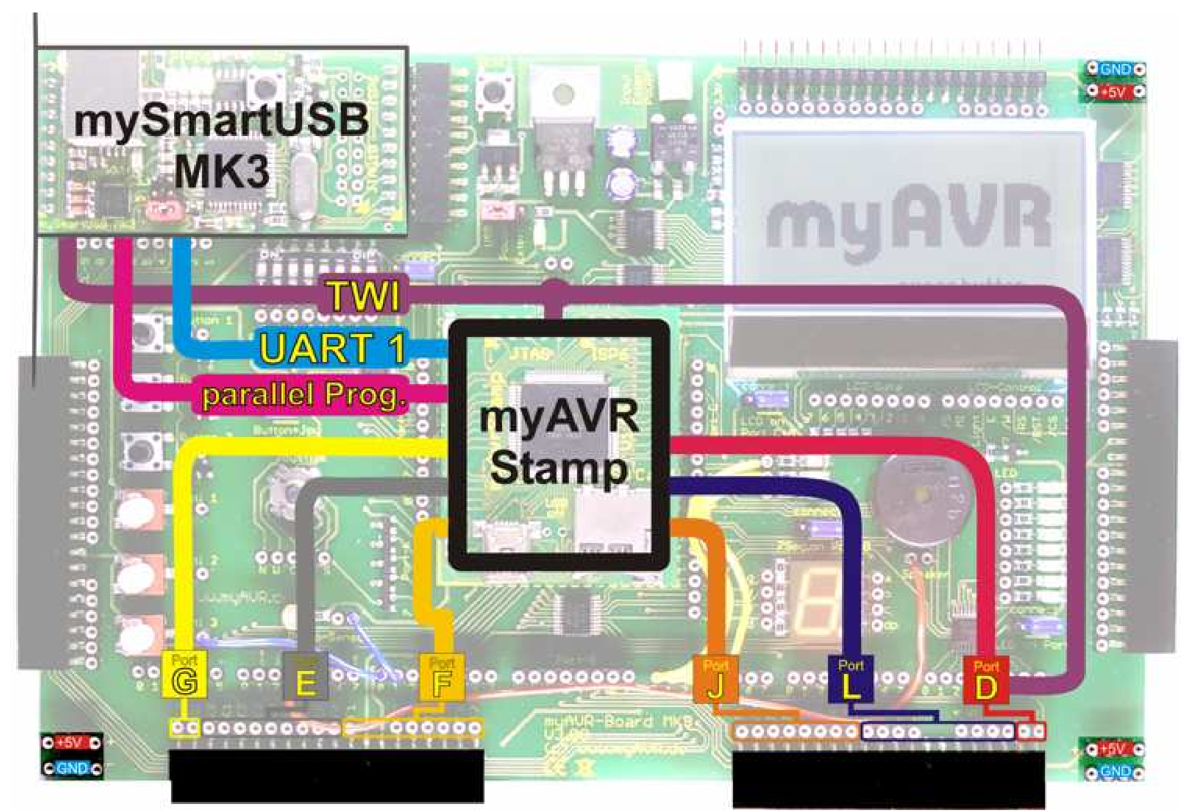


Abbildung: verfügbare Verbindungen des myAVR Board MK3

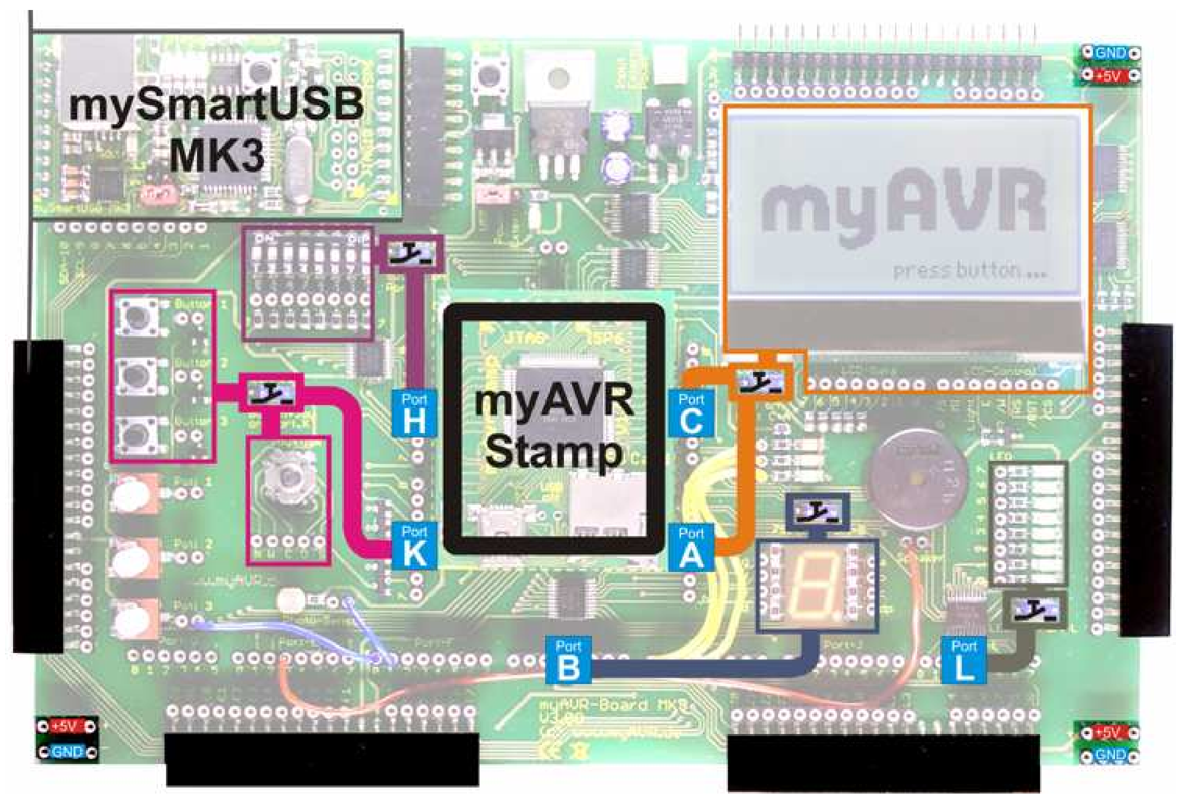


Abbildung: quick connect Verbindungen des myAVR Board MK3

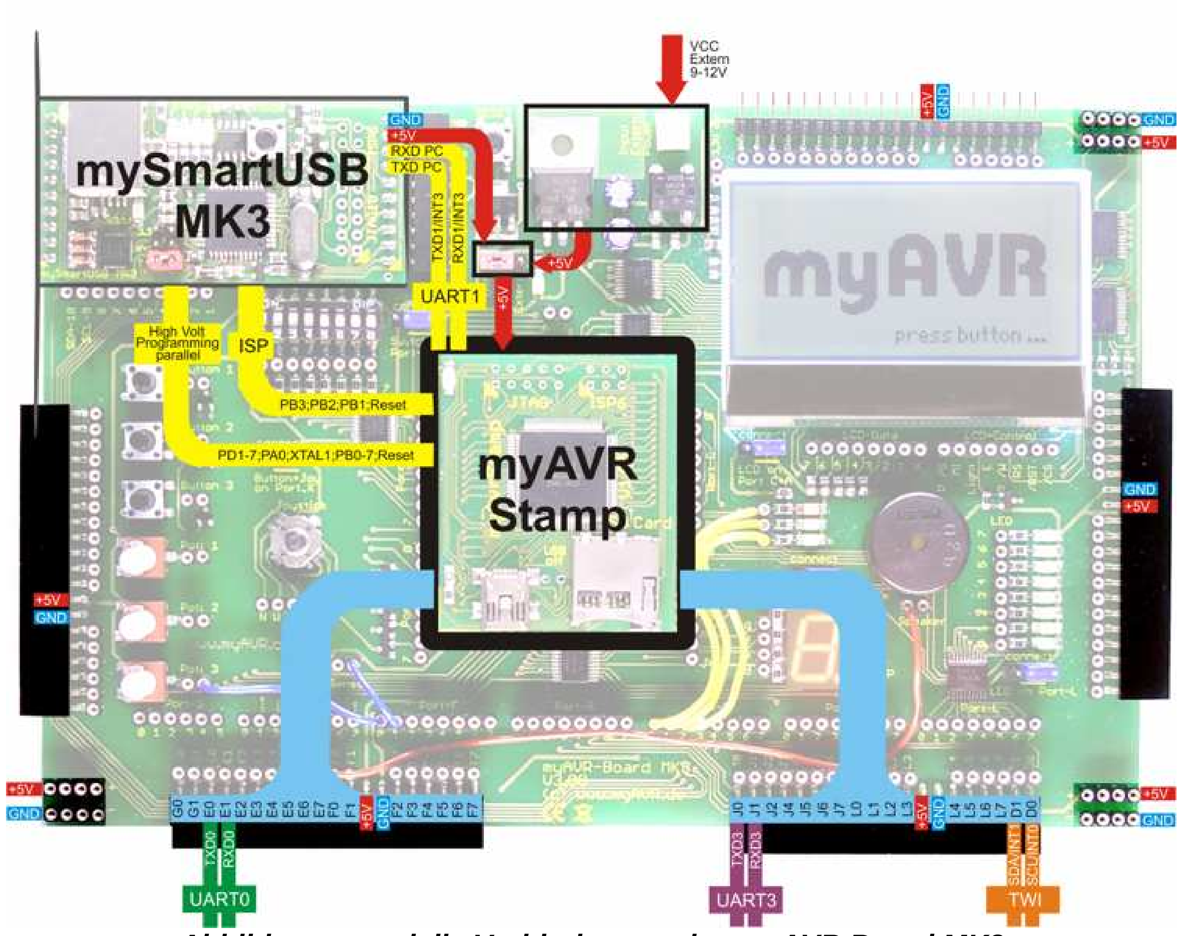
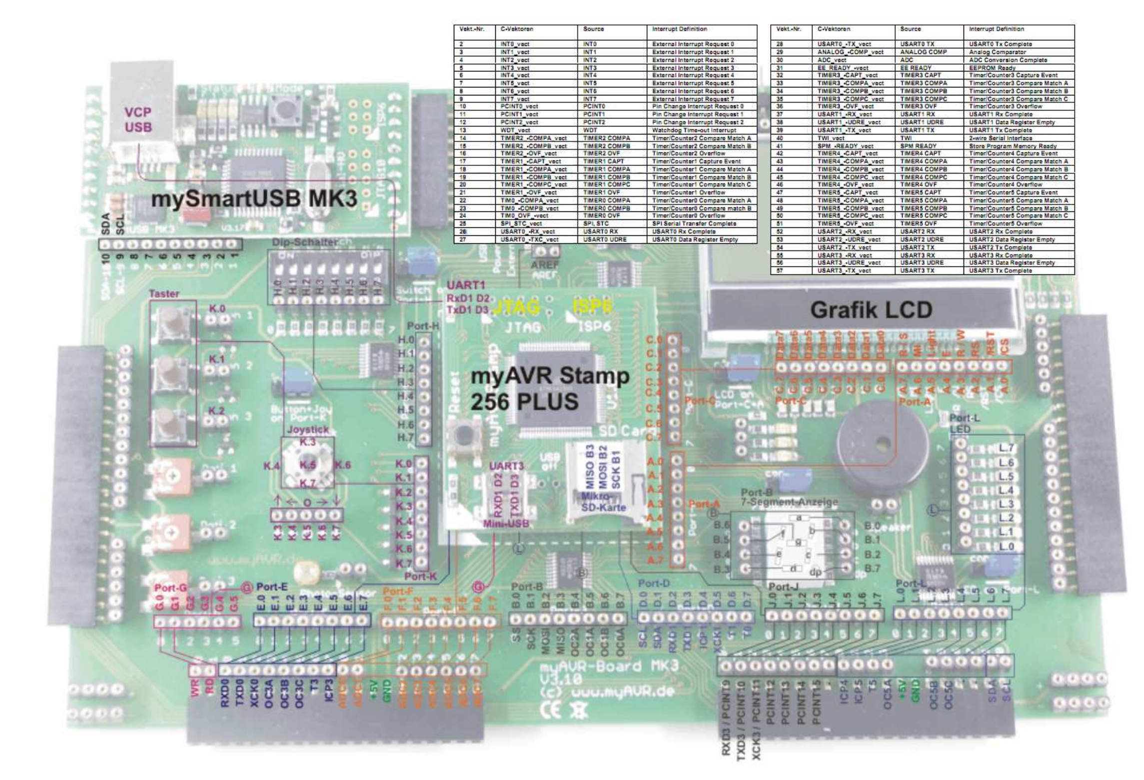


Abbildung: spezielle Verbindungen des myAVR Board MK3



Vekt.-Nr.	C-Vektoren	Source	Interrupt Definition
2	INT0_vect	INT0	External Interrupt Request 0
3	INT1_vect	INT1	External Interrupt Request 1
4	INT2_vect	INT2	External Interrupt Request 2
5	INT3_vect	INT3	External Interrupt Request 3
6	INT4_vect	INT4	External Interrupt Request 4
7	INT5_vect	INT5	External Interrupt Request 5
8	INT6_vect	INT6	External Interrupt Request 6
9	INT7_vect	INT7	External Interrupt Request 7
10	PCINT0_vect	PCINT0	Pin Change Interrupt Request 0
11	PCINT1_vect	PCINT1	Pin Change Interrupt Request 1
12	PCINT2_vect	PCINT2	Pin Change Interrupt Request 2
13	WDT_vect	WDT	Watchdog Time-out Interrupt
14	TIMER2_-COMPA_vect	TIMER2 COMPA	Timer/Counter2 Compare Match A
15	TIMER2_-COMPB_vect	TIMER2 COMPB	Timer/Counter2 Compare Match B
16	TIMER2_-OVF_vect	TIMER2 OVF	Timer/Counter2 Overflow
17	TIMER1_-CAPT_vect	TIMER1 CAPT	Timer/Counter1 Capture Event
18	TIMER1_-COMPA_vect	TIMER1 COMPA	Timer/Counter1 Compare Match A
19	TIMER1_-COMPB_vect	TIMER1 COMPB	Timer/Counter1 Compare Match B
20	TIMER1_-COMPC_vect	TIMER1 COMPC	Timer/Counter1 Compare Match C
21	TIMER1_-OVF_vect	TIMER1 OVF	Timer/Counter1 Overflow
22	TIM0_-COMPA_vect	TIMER0 COMPA	Timer/Counter0 Compare Match A
23	TIM0_-COMPB_vect	TIMER0 COMPB	Timer/Counter0 Compare match B
24	TIM0_OVF_-vect	TIMER0 OVF	Timer/Counter0 Overflow
25	SPI_STC_vect	SPI, STC	SPI Serial Transfer Complete
26	USART0_-RX_vect	USART0 RX	USART0 Rx Complete
27	USART0_-TXC_vect	USART0 UDRE	USART0 Data Register Empty
28	USART0_-TX_vect	USART0 TX	USART0 Tx Complete
29	ANALOG_-COMP_vect	ANALOG COMP	Analog Comparator
30	ADC_vect	ADC	ADC Conversion Complete
31	EE_READY_-vect	EE READY	EEPROM Ready
32	TIMER3_-CAPT_vect	TIMER3 CAPT	Timer/Counter3 Capture Event
33	TIMER3_-COMPA_vect	TIMER3 COMPA	Timer/Counter3 Compare Match A
34	TIMER3_-COMPB_vect	TIMER3 COMPB	Timer/Counter3 Compare Match B
35	TIMER3_-COMPC_vect	TIMER3 COMPC	Timer/Counter3 Compare Match C
36	TIMER3_-OVF_vect	TIMER3 OVF	Timer/Counter3 Overflow
37	USART1_-RX_vect	USART1 RX	USART1 Rx Complete
38	USART1_-UDRE_vect	USART1 UDRE	USART1 Data Register Empty
39	USART1_-TX_vect	USART1 TX	USART1 Tx Complete
40	TWI_vect	TWI	2-wire Serial Interface
41	SPM_-READY_vect	SPM READY	Store Program Memory Ready
42	TIMER4_-CAPT_vect	TIMER4 CAPT	Timer/Counter4 Capture Event
43	TIMER4_-COMPA_vect	TIMER4 COMPA	Timer/Counter4 Compare Match A
44	TIMER4_-COMPB_vect	TIMER4 COMPB	Timer/Counter4 Compare Match B
45	TIMER4_-COMPC_vect	TIMER4 COMPC	Timer/Counter4 Compare Match C
46	TIMER4_-OVF_vect	TIMER4 OVF	Timer/Counter4 Overflow
47	TIMER5_-CAPT_vect	TIMER5 CAPT	Timer/Counter5 Capture Event
48	TIMER5_-COMPA_vect	TIMER5 COMPA	Timer/Counter5 Compare Match A
49	TIMER5_-COMPB_vect	TIMER5 COMPB	Timer/Counter5 Compare Match B
50	TIMER5_-COMPC_vect	TIMER5 COMPC	Timer/Counter5 Compare Match C
51	TIMER5_-OVF_vect	TIMER5 OVF	Timer/Counter5 Overflow
52	USART2_-RX_vect	USART2 RX	USART2 Rx Complete
53	USART2_-UDRE_vect	USART2 UDRE	USART2 Data Register Empty
54	USART2_-TX_vect	USART2 TX	USART2 Tx Complete
55	USART3_-RX_vect	USART3 RX	USART3 Rx Complete
56	USART3_-UDRE_vect	USART3 UDRE	USART3 Data Register Empty
57	USART3_-TX_vect	USART3 TX	USART3 Tx Complete

Abbildung: detaillierte Belegungen am myAVR Board MK3

Der auf dem Board befindliche Controller gehört zur Reihe megaAVR und verfügt über die maximale Ausstattung an möglichen Baugruppen. Die folgenden Darstellungen aus dem Datenblatt des ATmega2560 sollen einen Überblick zu dessen Möglichkeiten geben.

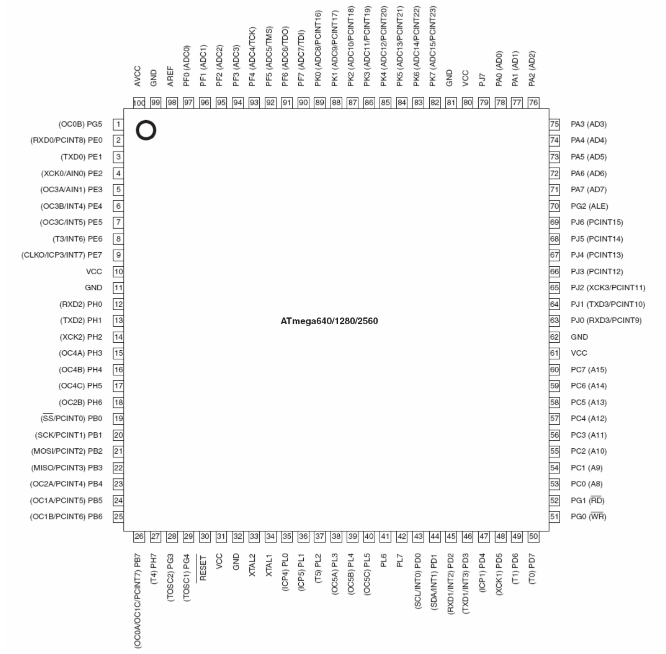


Abbildung: Pin-Out des ATMega2560

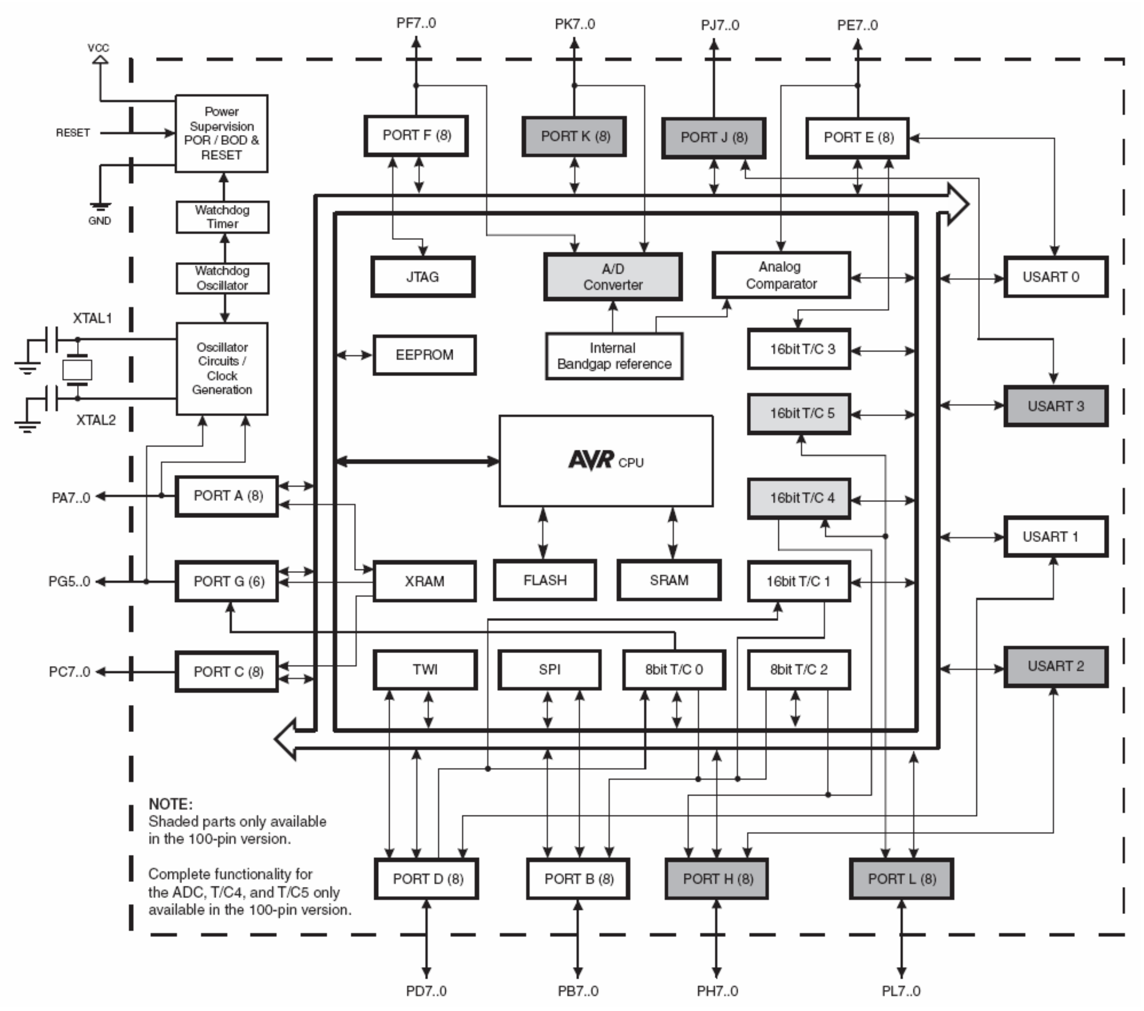


Abbildung: Aufbau des AVR RISC Controllers ATmega2560 der Firma Atmel

Inbetriebnahme der Ziel-Hardware

Stellen Sie die Betriebsbereitschaft Ihres myAVR-Boards her. Dazu muss die vorher beschriebene Installation der Hardwaretreiber erfolgt sein. Bitte verbinden Sie das Board mit dem PC über das USB-Kabel. Die USB-Hardware wird vom Betriebssystem erkannt und installiert.



Abbildung: Das erste Anstecken des myAVR Board

Das Board ist jetzt als virtueller COM-Port (VCP) im System verfügbar. Sie können die vom Betriebssystem zugewiesene Nummer des Ports im Gerätemanager überprüfen bzw. auch ändern.

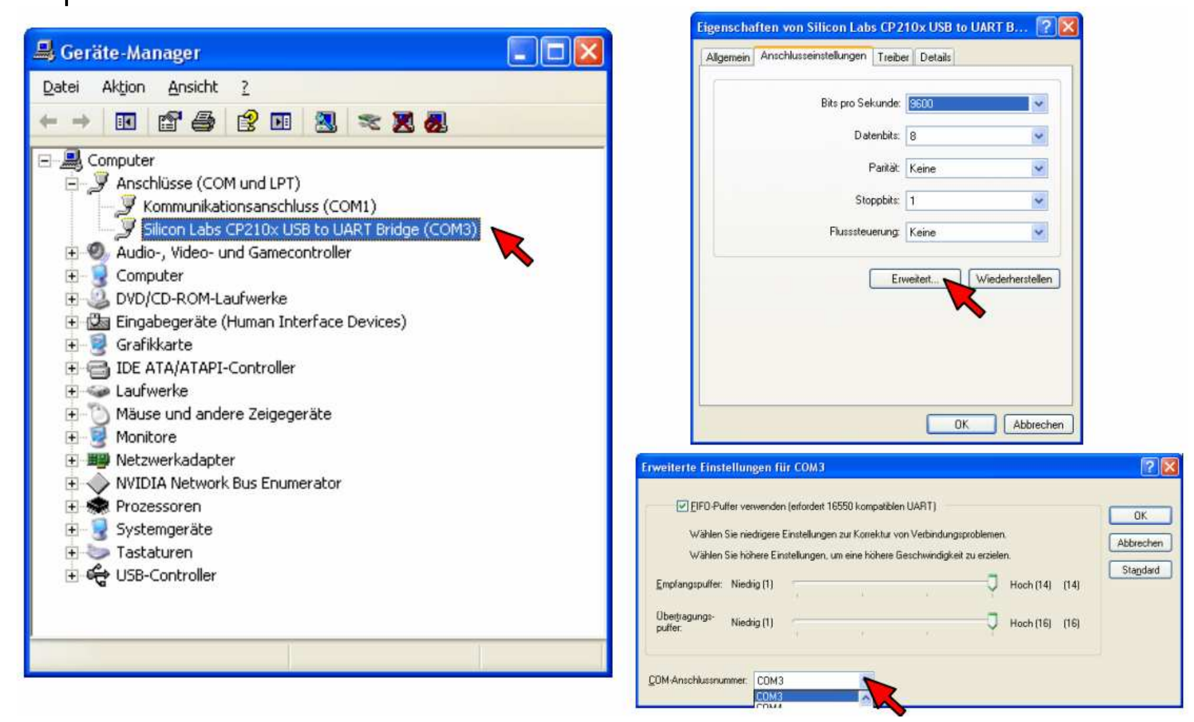
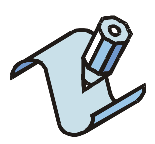


Abbildung: Das myAVR Board im Gerätemanager



Aufgabe:
Führen Sie den Systemtest an Ihrem myAVR-Board MK3 aus und protokollieren Sie das Ergebnis.

Hinweis:
Die folgende Testliste enthält ausgewählte Beispiele aus dem gesamten Testprogramm. Für einen Test mit der SD-CARD benötigen Sie eine FAT16-formatierte SD-Card mit Dateien in der Wurzel.

Boardtest

- Das Testprogramm erscheint beim Start.
  ☐ JA ☐ NEIN
- Zum nächsten Testschritt gelangen Sie mit Drücken von Taster 3
  ☐ JA ☐ NEIN
- Zum Start gelangen Sie zurück, wenn Sie Taster 1 drücken
  ☐ JA ☐ NEIN
- Testverkabelung entsprechend Bild im Abschnitt „Die Referenzhardware“
  ☐ JA ☐ NEIN

Start:

- Hauptmenü wird im Grafik LCD erzeugt
  ☐ JA ☐ NEIN
- 7-Segment-Anzeige gibt Zahlen aus
  ☐ JA ☐ NEIN
- LEDs leuchten alle nacheinander
  ☐ JA ☐ NEIN

Weiter mit Taste 3:

- Ihnen erscheint das erste Font-Beispiel, verschiedene Texttypen sind zu erkennen
  ☐ JA ☐ NEIN
- Drücken Sie den Joystick, um sich ein weiteres Fontbeispiel anzusehen
  ☐ JA ☐ NEIN

Weiter mit Taste 3::

- In der nächsten Anzeige, die Ihnen erscheint, können Sie mit Hilfe des Joysticks den Cursor beliebig bewegen
  ☐ JA ☐ NEIN

Weiter mit Taste 3::

- Zur Änderung des Kurvenverlaufs im Analogmenü drehen Sie am untersten Potentiometer bzw. variieren den Lichteinfall auf den Lichtsensor
  ☐ JA ☐ NEIN

Weiter mit Taste 3::

- Schalten Sie nacheinander die DIP-Schalter ein
  ☐ JA ☐ NEIN
- Im Grafik LCD kontrollieren Sie die Anzeige der eingeschalteten DIP-Schalter
  ☐ JA ☐ NEIN

Weiter mit Taste 3::

- Viele verschiedene Grafiken erzeugen sich selbstständig
  ☐ JA ☐ NEIN

Menü

Start	Button 1	Home
Zurück	Button 2	Back
Weiter	Button 3	Next

myAVR
press button ...

Fontbeispiele
Textzeile 1
Textzeile 2
Textzeile 3
Textzeile 4
Joystick drücken ...

Joystick
in alle Richtungen bewegen
und Taste drücken

Analog

DIP-Schalter
1 2 3 4 5 6 7 8
DIPs nacheinander ein

Grafik

.....

2.4 Was ist SiSy?

SiSy ist die Abkürzung für Simple System. Dabei steht System dafür, dass Systeme egal ob klein, mittel oder groß strukturiert und methodisch mit standardisierten Darstellungsmitteln konstruiert werden. Simple steht für eine einfache Vorgehensweise und übersichtliche Darstellung. SiSy bildet die Darstellungsmittel zur Konstruktion eines Systems individuell und aufgabenspezifisch ab. Das bedeutet, dass für jede spezifische Konstruktionsaufgabe auch spezielle Darstellungstechniken zur Verfügung stehen. Die Art der mit SiSy zu konstruierenden Systeme kann sehr vielfältig sein. Die Einsatzmöglichkeiten reichen von der Konstruktion von Softwaresystemen für Mikrocontroller über Datenbanklösungen auf Arbeitsstationen oder Servern bis hin zu betriebswirtschaftlichen Managementsystemen. SiSy ist ein allgemeines Modellierungswerkzeug für beliebige Systeme.

2.5 Grundaufbau des Entwicklungswerkzeuges

Schauen wir uns als nächstes kurz in der Entwicklungsumgebung SiSy AVR um. SiSy AVR ist, wie bereits erwähnt, ein allgemeines Entwicklungswerkzeug, mit dem man von der Konzeption eines Systems, bis zur Realisierung die verschiedensten Arbeitsschritte unterstützen kann. Für die Eingabe von Programmcode mit oder ohne Modellen bzw. Diagrammen bietet SiSy als Basiskomponente einen Zeileneditor mit Syntaxfarben und Hilfefunktionen an. Modelle werden als Diagramme erstellt bzw. abgebildet.

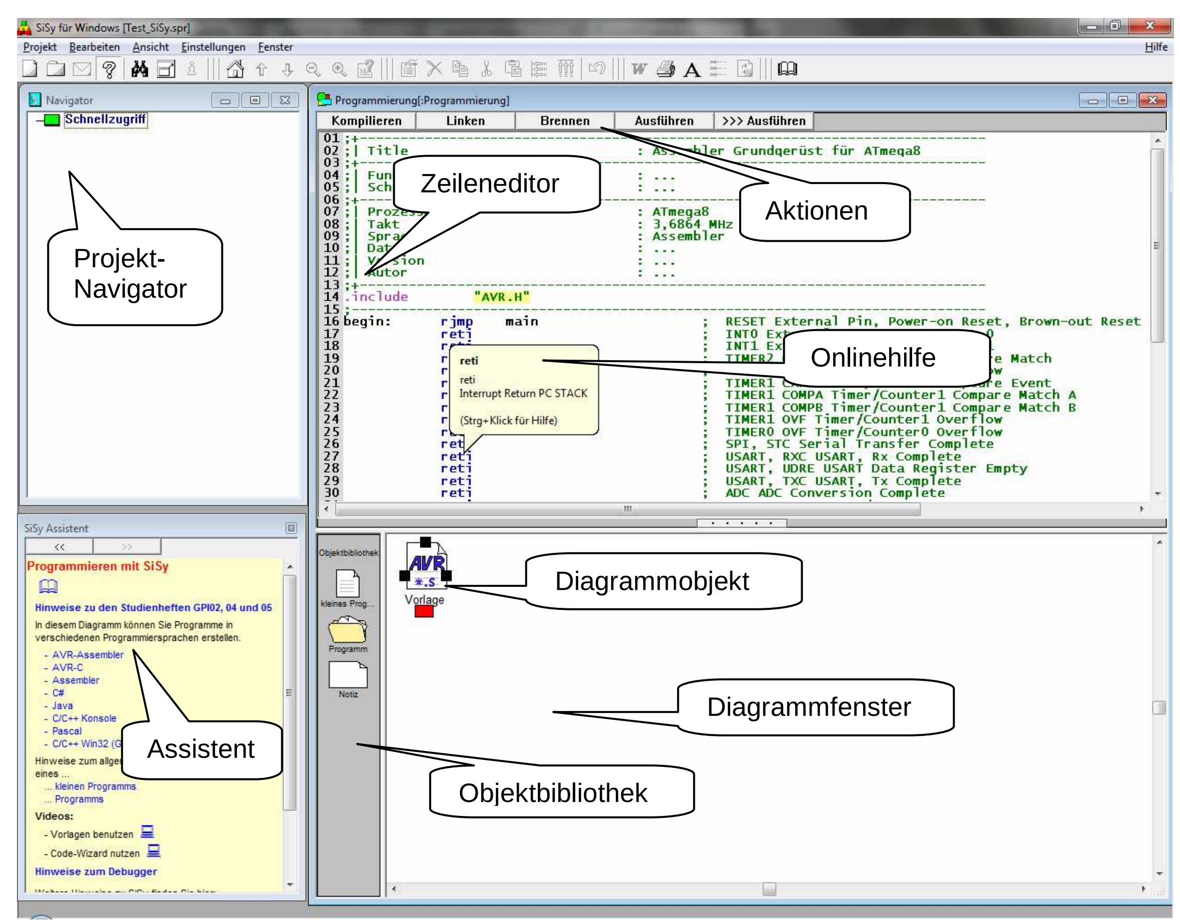


Abbildung: Bildschirmaufbau der Entwicklungsumgebung SiSy AVR

Beim Kompilieren, Linken oder auch Brennen öffnet sich ein Ausgabefenster und zeigt Protokollausgaben der Aktionen an. Wenn die Hardware ordnungsgemäß angeschlossen, von der Software erkannt und das Programm erfolgreich übersetzt

sowie auf den Programmspeicher des Mikrocontrollers übertragen wurde, muss die letzte Ausschrift in Abhänigkeit der Konfiguration folgenden bzw. ähnlichen Inhalt haben:

```
vorbereiten ...
brennen ...
benutze: mySmartUSB MK2 an COM2 mit ATmega8
USB-Treiber installiert, aktiv (V ), Port: COM2
Prozessor: ATmega8
schreibe 50 Bytes in Flash-Memory ...
... erfolgreich (0.32 s)
OK
```

Abbildung: ProgTool Ausgabefenster mit „Brenn“ - Protokoll

Die Inbetriebnahme, Test und Datenkommunikation mit der Mikrocontrollerlösung erfolgen über das myAVR-Controlcenter. Dabei wird über die Schaltfläche „Start“ das Testboard mit der nötigen Betriebsspannung versorgt und der Controller gestartet. Der Datenaustausch mit dem myAVR Board ist möglich, wenn das Null-Modemkabel (oder USB-Kabel) an Rechner und Testboard angeschlossen ist, sowie die Mikrocontrollerlösung dafür vorgesehen ist. Es können Texte und Bytes (vorzeichenlose ganzzahlige Werte bis 255) an das Board gesendet und Text empfangen werden. Die empfangenen Daten werden im Protokollfenster angezeigt.

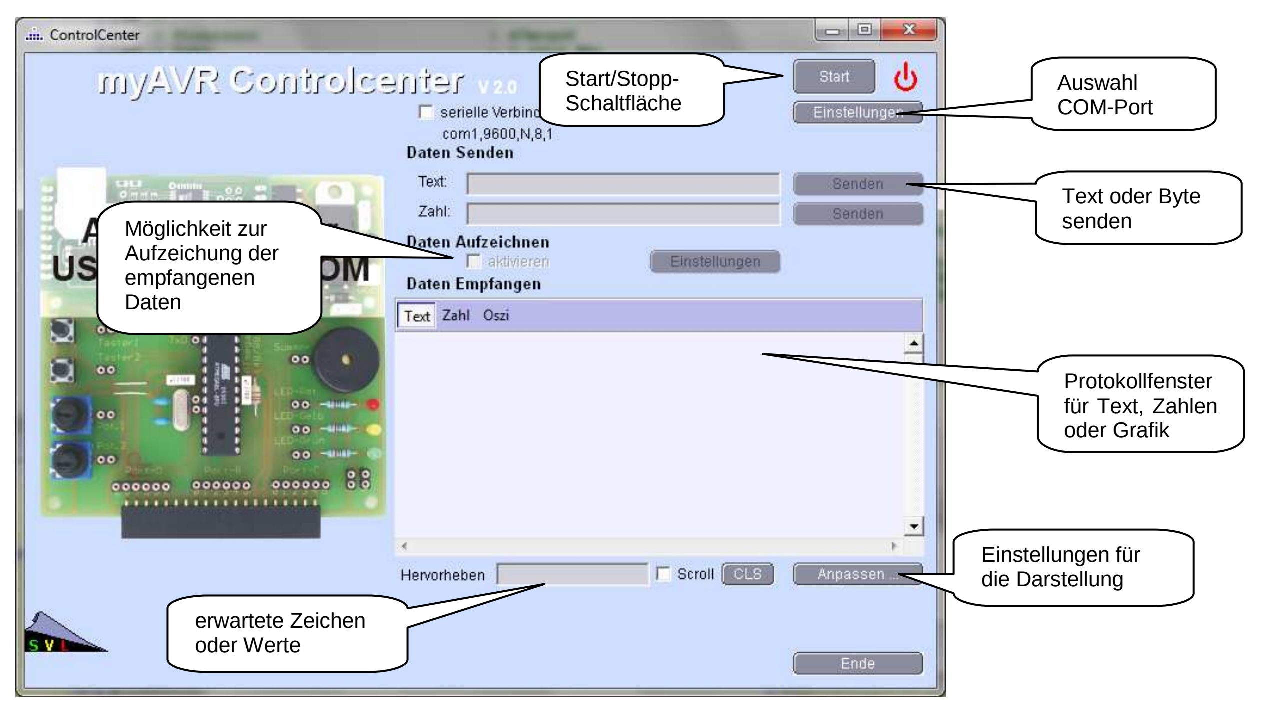


Abbildung: myAVR-Controlcenter

Nutzen Sie die zahlreichen Hilfen und Vorlagen, die SiSy AVR bietet!

Der Assistent von SiSy bietet Ihnen Beispielprogramme, Hinweise und interessante Lösungen. Eine ausführliche Beschreibung zum Assistenten und der Hilfefunktionen, z.B. Syntax zu Befehlen oder Druckmöglichkeiten, finden Sie im Benutzerhandbuch von SiSy-AVR

2.6 Grundlagen der Bedienung von SiSy

Sollten Sie mit der allgemeinen Bedienung von SiSy bereits vertraut sein können Sie dieses Kapitel überspringen. Im Benutzerhandbuch und mit den verschiedenen Schnelleinstiegen im Downloadbereich unter www.myAVR.de finden Sie zahlreiche Anleitungen und Anregungen zu den Möglichkeiten des Werkzeuges SiSy. Im Folgenden sollen für die weitere Arbeit immer wiederkehrende Handhabungsschritte kurz erklärt werden.

Ein neues Projekt anlegen

Starten Sie SiSy und wählen „Neues Projekt erstellen“. Legen Sie einen Namen für das Projekt fest. Es wird ein Projektverzeichnis erzeugt in dem die zum Projekt gehörenden Dateien abgelegt werden. Es kann in einem Projektverzeichnis immer nur eine Projektdatenbank enthalten sein.

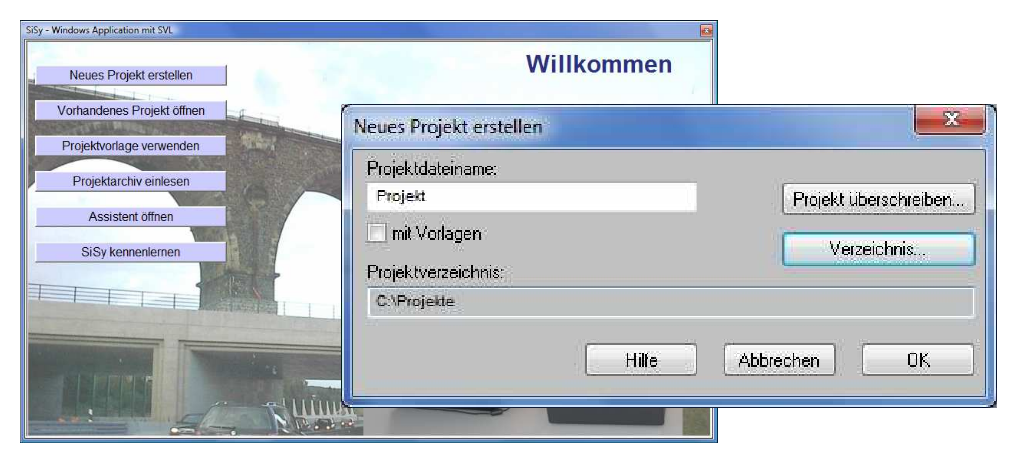


Abbildung: ein neues SiSy-Projekt erstellen

Für die weitere Arbeit ist es nötig, sich auf ein so genanntes Vorgehensmodell festzulegen. SiSy arbeitet diagrammorientiert. Das bedeutet, dass alle Elemente die zu einem Projekt gehören als grafische Objekte in einem Diagramm angelegt werden. Bestimmte Elemente können wiederum wie ein Hyperlink mit Diagrammen hinterlegt werden. Das Vorgehensmodell ist die Einstiegsebene für die zu erstellenden Diagramme. Es bildet das Home-Verzeichnis von dem aus alle weiteren Diagramme zu erreichen sind. Das Vorgehensmodell legt auch fest, welche Diagrammtypen im Projekt zur Verfügung stehen. Wählen Sie für diese Übung das AVR-Vorgehensmodell aus.

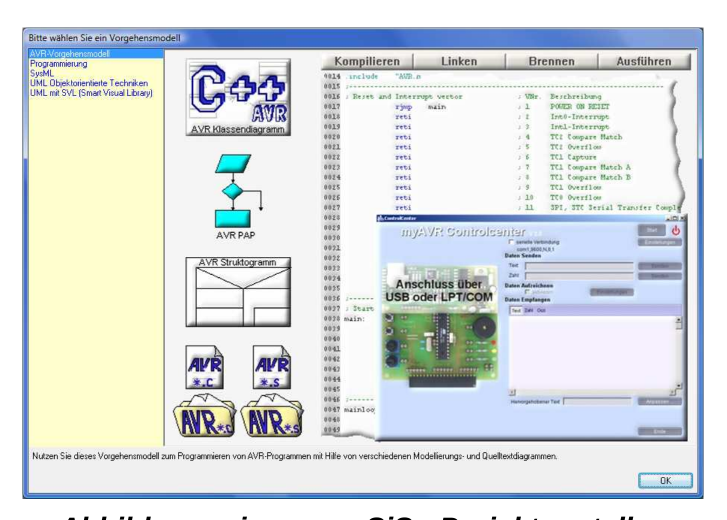


Abbildung: ein neues SiSy-Projekt erstellen

Einstellungen für die Zielhardware

Bei der Entwicklung von Mikrocontrollerlösungen wird auf dem PC der Quellcode geschrieben und in Maschinencode übersetzt. Der Maschinencode ist jedoch nicht auf dem PC ausführbar, sondern muss in den Programmspeicher des Mikrocontrollers übertragen werden. Dazu muss die Zielhardware über ein Programmiergerät an den PC angeschlossen werden. In unserem Fall ist das Programmiergerät der mySmartUSB MK3. Dieser ist auf dem MK3 Board als Tochterplatine integriert. Der Mikrocontroller befindet sich auf der myAVR Stamp. Im folgenden Beispiel wurde ein MK3 Board mit einer myAVR Stamp 256 Plus verwendet. Diese ist mit einem ATmega2560 bestückt. Dieser wird von einem 16 MHz Quarz getaktet. SiSy fordert Sie bei der Wahl des AVR-Vorgehensmodells dazu auf, die Hardwareinformationen für das gesamte Projekt festzulegen.



Abbildung: der Hardware-Assitent wird gestartet

Wählen Sie im Einstellungsdialog für die Hardware, den mySmartUSB MK3. Ermitteln Sie über die zugehörige Schaltfläche mit dem Fragezeichen den vom Betriebssystem festgelegten Port und übernehmen Sie die Einstellung. Durch wiederholtes Betätigen der Schaltfläche können Sie den angeschlossenen Controller ermitteln lassen. Übernehmen Sie auch hier die korrekten Einstellungen für den Controllertyp. Speichern Sie die vorgenommenen Änderungen.

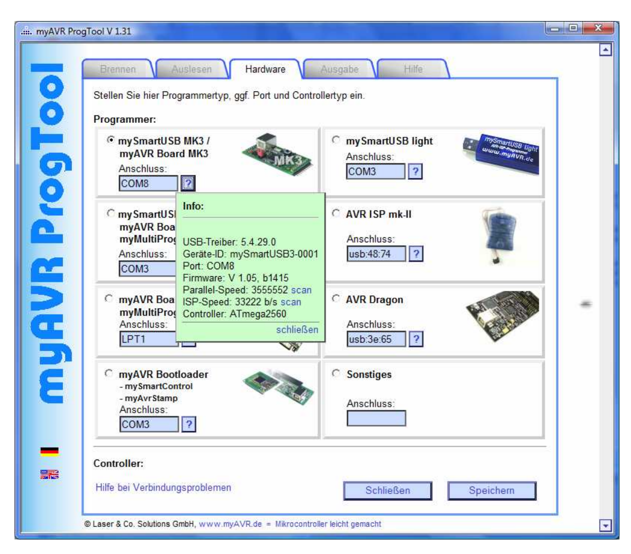


Abbildung: Die Einstellungen für das MK3 Board wurden ermittelt

Für eine Vielzahl von Funktionen ist es wichtig mit welcher Geschwindigkeit der Controller läuft. Die Geschwindigkeit des Controllers wird durch seinen Takt bestimmt. Das ist eine von der vorliegenden Hardware bestimmte Größe und muss

unbedingt in der Software mit den tatsächlichen Hardwaregegebenheiten übereinstimmen. Im Auslieferungszustand ist die MK3 Hardware auf den 16 MHz Quarz der myAVR Stamp eingestellt. Diese Taktgeschwindigkeit muss für das angelegte Projekt ausgewählt werden. Mit Betätigen der Schaltfläche „Fertigstellen“ werden die gewählten Einstellungen als Projektstandards gespeichert.

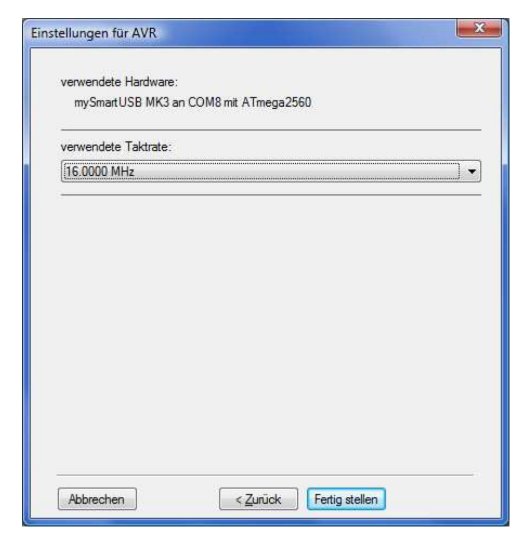


Abbildung: Die Taktrate für das MK3 Board wurden eingestellt

Vorlagen nutzen

SiSy bietet je nach Vorgehensmodell und gewählten Optionen geeignete Vorlagen mit Bibliotheken und Beispielen. Für die ersten Schritte wählen wir fertige Beispiele um diese zu testen und die Arbeit mit SiSy kennenzulernen. Wählen Sie die Diagrammvorlage „AVR C++ Bibliotheken und Beispiele“. Es wird ein komplettes Projekt mit UML Klassenbibliotheken und Anwendungsbeispielen importiert.

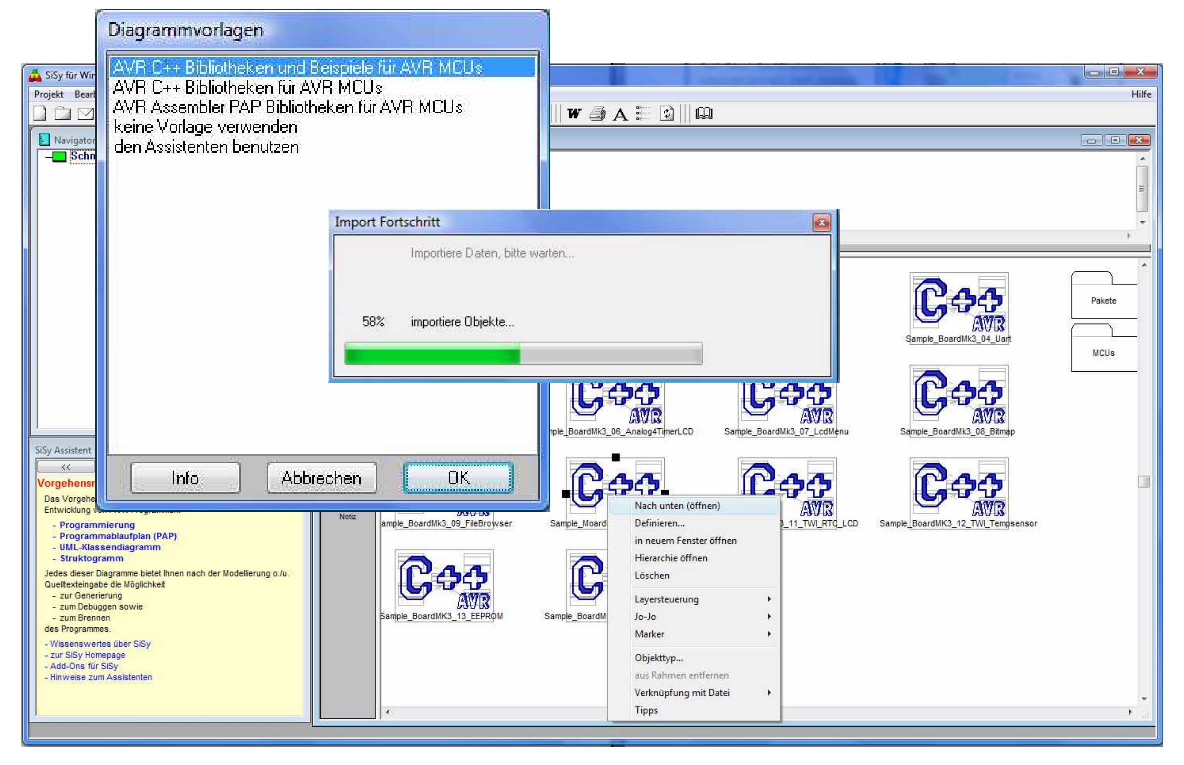


Abbildung: Eine Diagrammvorlage mit Beispielen laden

Jedes Symbol in dem jetzt vorliegenden Diagramm enthält wiederum ein Diagramm. Um in eines der Diagramme zu gelangen, selektieren Sie das Symbol und

wählen im Kontextmenü (rechte Maustaste auf dem Symbol) den Menüpunkt „nach unten (öffnen)“.

Für diese Übung nutzen Sie bitte das Beispiel „Saple_BoardMk3_10_GamePing“. Öffnen Sie das Diagramm!

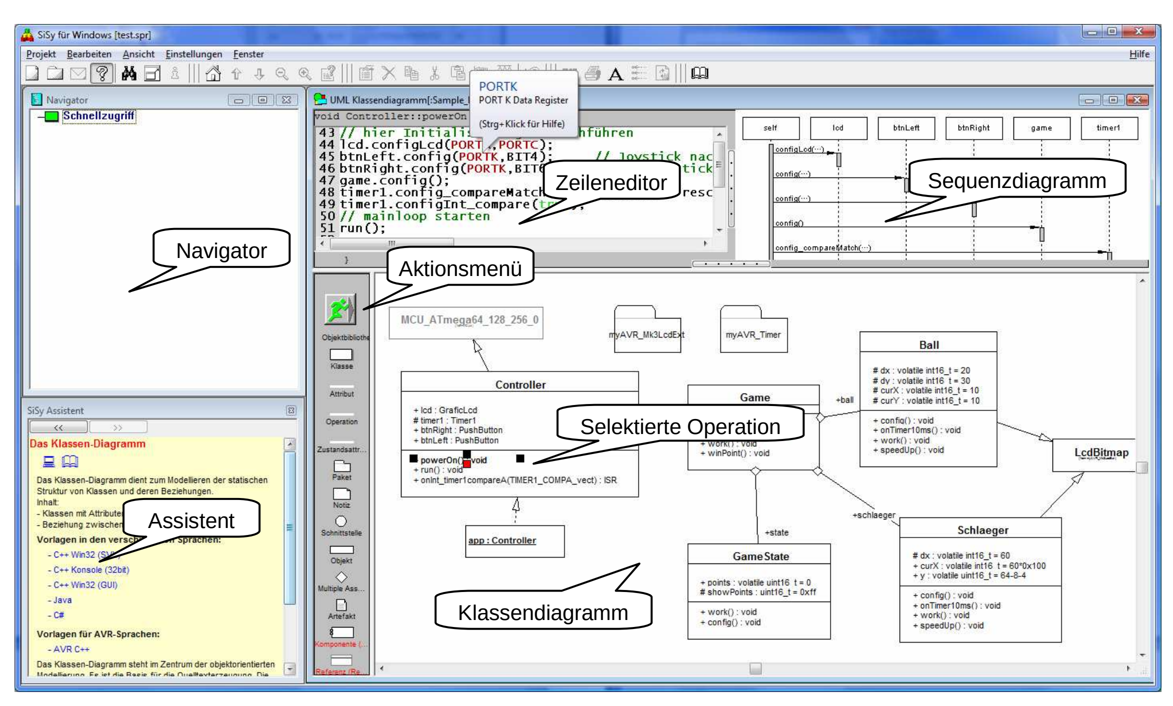


Abbildung: Ein komplettes UML Klassendiagramm

Eine Besonderheit im Klassendiagramm ist, dass für eine Operation, die selektiert wird, aus dem im Zeileneditor angezeigten Text automatisch das entsprechende Sequenzdiagramm generiert wird.

Programme übersetzen und brennen

Um das Beispiel zu testen ist es nötig, dieses zu Erstellen (Quellcode generieren, Kompilieren, Linken) und auf das MK3 Board zu übertragen (Brennen). Diese Aktionen erreichen Sie über das Aktionsmenü.

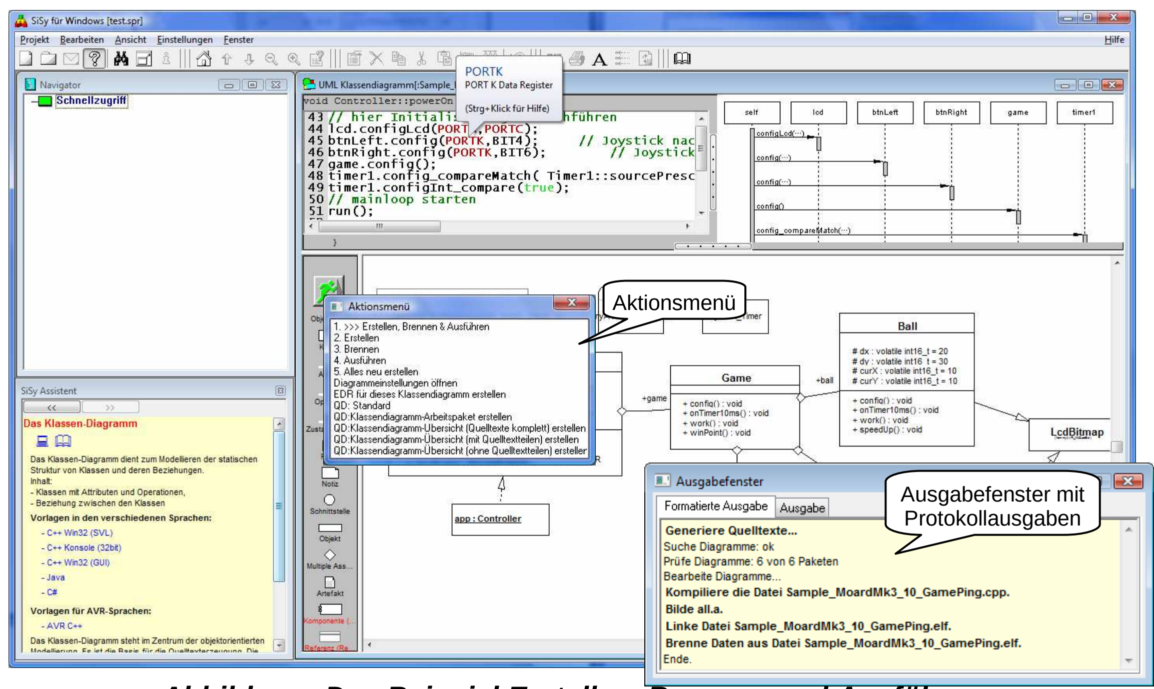


Abbildung: Das Beispiel Erstellen, Brennen und Ausführen

Für das Übertragen (Brennen) auf den Mikrocontroller öffnet sich das myAVR ProgTool. Es ist für den gesamten Vorgang des Brennens und das Überprüfen der erfolgreichen Übertragung verantwortlich. Dafür verfügt es über ein eigenes Protokollfenster. Treten Fehler bei der Übertragung auf, bleibt das Protokollfenster offen und ist rot hinterlegt.

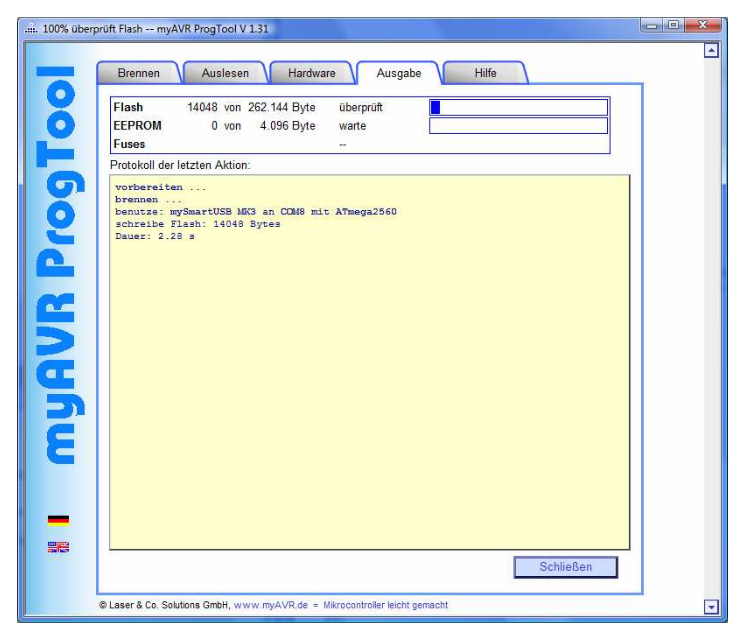


Abbildung: Das myAVR ProgTool nach erfolgreichem Brennen

Nach dem Erstellen und Brennen wird die Anwendung ausgeführt. Das MK3 Board erhält die nötige Spannungsversorgung über den USB Anschluss und Sie können das Beispiel testen. Beachten Sie, dass die Button und das Display aktiviert, also mit dem Controller verbunden sind (Quick Connect Option).

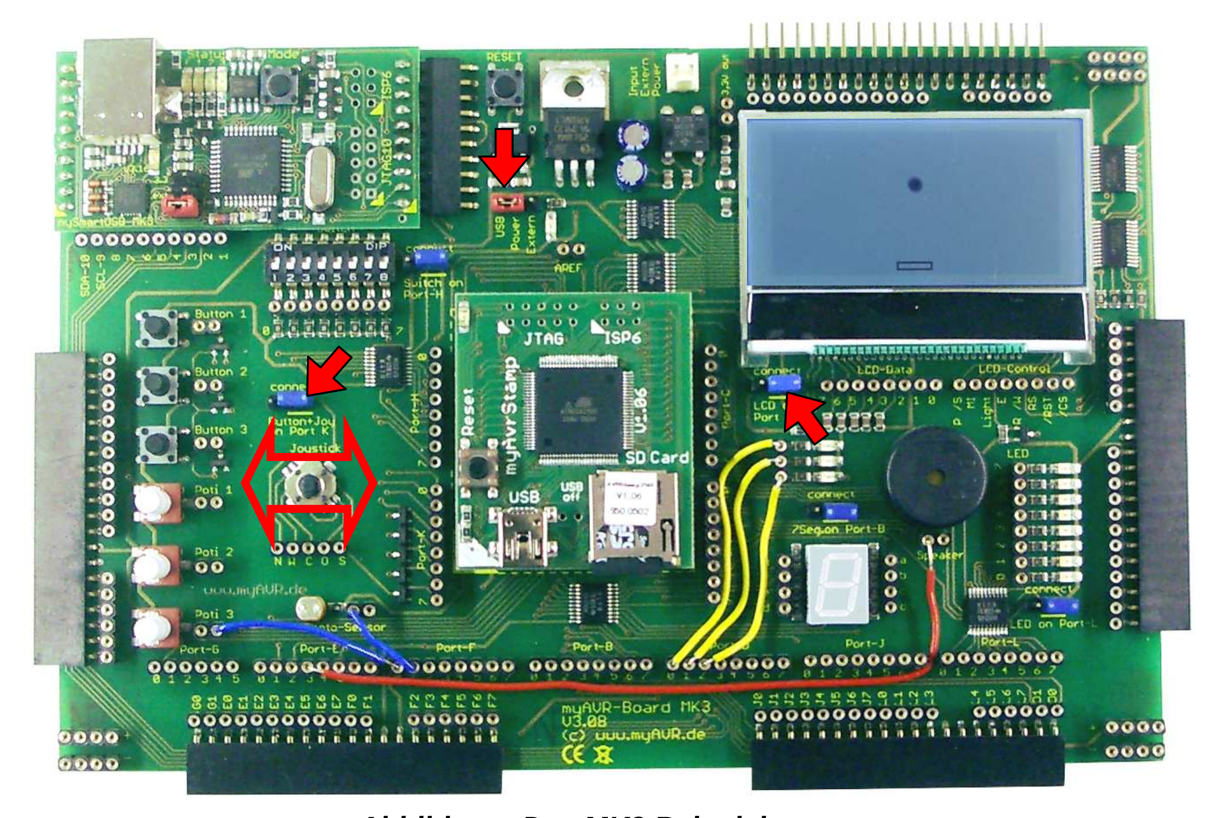
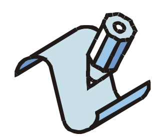


Abbildung: Das MK3 Beispiel testen

Mit SiSy ab der Ausgabe AVR++ bzw. den entsprechenden Add-Ons können aber nicht nur AVR Mikrocontroller programmiert werden, sondern auch PC Anwendungen. Sie sollten mit den soeben erworbenen Kenntnissen in der Lage sein, ein fertiges Beispiel für eine Windowsanwendung zu testen.



Aufgabe:
Testen Sie die Möglichkeiten, von SiSy Windowsanwendungen zu erstellen. Nutzen Sie verfügbare Vorlagen.

Hinweise:

- Legen Sie ein neues Projekt an.
- Wählen Sie das Vorgehensmodell „UML mit SVL (Smart Visual Library)“.
- Laden Sie die Diagrammvorlage „SVL (Smart Visual Library)“.
- Öffnen sie das Klassendiagramm „Ballspiel“.
- Kompilieren, Linken und Starten Sie das Beispiel.



Abbildung: Das SVL Beispiel testen

3 Kleine Systeme konstruieren

*„Wer sich zu groß fühlt, um kleine Aufgaben zu erfüllen,
ist zu klein, um mit großen Aufgaben betraut zu werden.“*
Jacques Tati



Abbildung: ein ATtiny10 (Bild Atmel)

Wer glaubt, dass Assemblerprogrammierung ein alter Hut ist sollte sich ein wenig mit den jüngsten, kleinsten und energieeffizientesten tinyAVR beschäftigen. Diese besitzen erstaunliche Eigenschaften. Abmessungen von 2 mm x 2 mm und eine möglichen Versorgungsspannung von unter 1 Volt, einen Verbrauch im Mikroampere-Bereich und weniger, trotzdem bis zu 12 Millionen Operationen pro Sekunde aber eben auch nur 32 Byte Arbeitsspeicher (in Worten zweiunddreißig). Hochsprachen wie C oder C++ verbrauchen viel zu viel dieser knappen Ressource der genialen kleinen Zwerge.

3.1 Die von-Neumann-Architektur und Sprungorientierung

In den späten 1940er Jahren leistete der Ungar János Neumann (John von Neumann 1903-1957) aus seiner Tätigkeit zur Stabilitätsanalyse numerischer Rechenverfahren beim Manhattenprojekt heraus einen maßgeblichen Beitrag zum Bau der ersten amerikanischen Universalrechner. Er war in dieser Beziehung sozusagen der amerikanische Conrad Zuse (1910-1995). Auch wenn er nicht der Schöpfer des ersten funktionsfähigen Digitalrechners war, so geht jedoch auf ihn die Systematisierung der seinerzeit wichtigsten Prinzipien für die Funktionsweise und des Aufbaus von digitalen Universalrechnern zurück. Man bezeichnet diese Systematisierung als *von-Neuman-Architektur*. Rechner, die nach diesen Prinzipien aufgebaut sind, nennt man von-Neumann-Rechner. Es wird sogar behauptet, er habe diese Systematisierung eigentlich als Patentschrift verfasst; diese sei aber aufgrund der damals bereits seit über 100 Jahren vorliegenden Patentschriften von Charles Babbage (1791-1871) als Patent abgelehnt worden. Vieles von dem was Herr von Neumann damals als revolutionäre Architekturprinzipien zusammenfasste, erscheint aus heutiger Sicht fast selbstverständlich. Damals waren aber zum Beispiel die heute verschwunden geglaubten Analogrechner bereits in der Massenanwendung zum Beispiel bei den Raketensteuerungen des Herrn von Braun. Analogrechner behaupteten sich noch lange gegen die Ablösung durch digitale Steuerungen. Digitalrechner galten damals noch lange nicht als der Stein der Weisen und wer weiß, wann man über Digitalrechner wie über Faustkeile schmunzelnd den Kopf schütteln wird. Vielleicht erleben wir das noch.

Zu den Prinzipien einer *von-Neumann-Architektur* gehören unter anderem:

- ein Rechner besteht mindestens aus Rechenwerk, Steuerwerk, Speicher, Ein- und Ausgabegeräten
- interne Informationen (Befehle und Daten) werden binär gespeichert (Digitalrechner)
- Befehle und Daten befinden sich in einem einheitlichen Speicher (Hauptspeicher, Arbeitsspeicher)

- der Speicher besteht aus Worten fester Länge (Speicherwort, Verarbeitungsbreite)
- der Speicher wird fortlaufend adressiert
- Befehle und Daten werden über ihre Position im Speicher angesprochen (Adressierung)
- die Befehle werden in der Reihenfolge ihrer Speicherung abgearbeitet (bis hier nur ein sequentieller Automat)
- von der sequentielle Verarbeitung kann durch unbedingte und bedingte Sprünge abgewichen werden (jetzt ein Turing Automat)

Und genau im letzten Punkt widerspiegelt sich das Programmierkonzept einer von-Neumann-Maschine. Die Befehle im Speicher werden nacheinander Schritt für Schritt abgearbeitet. Die eigentliche Programmlogik entsteht durch das Anwenden von Sprunganweisungen, welche über das Abweichen von der sequentiellen Verarbeitung den Algorithmus formen. Des Weiteren ergeben sich aus den genannten Punkten auch die für eine solche Maschine nötigen Befehlsgruppen. Befehle die Speicherinhalte bewegen um die Daten für Operationen bereitzustellen (Transportbefehle), Befehle für die eigentlichen Operationen welche also Berechnungen und Veränderungen der Daten ausführen (Arithmetik- und Logikbefehle), Befehle für das Ausführen von Sprüngen um den Algorithmus zu formen (Sprungbefehle). Der Sprung (*Jump, GoTo*) ist das Basiskonzept der Programmierung.

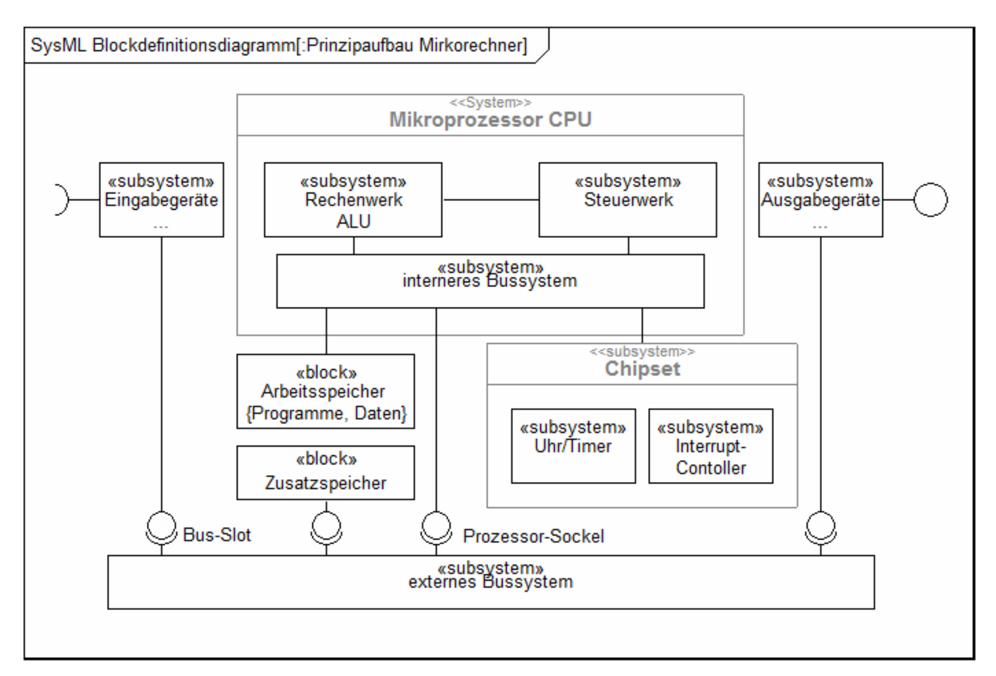


Abbildung: Prinzipaufbau eines Mikrorechners mit von Neumann Architektur

Die Zentraleinheit eines Mikrorechners, die Central Processing Unit (CPU) setzt sich zusammen aus dem Steuerwerk und dem Rechenwerk. Alle anderen Bausteine sind extern. Die Verbindung zwischen CPU und den benötigten Bausteinen erfolgt über ein externes Bussystem. Die komplette Zentraleinheit wird als Mikroprozessor bezeichnet. Wenn nun alle wesentlichen Baugruppen, die Zentraleinheit, das Speicherwerk und entsprechende Peripheriebausteine, in einem Gehäuse integriert sind, bezeichnet man diesen kleinen Komplettrechner als Mikrocontroller, Ein-Chip-Mikrorechner oder Embedded Computer. Dabei wird oft das Prinzip der Harvard Architektur angewendet. Damit bezeichnet man die im Gegensatz zur von Neumann Architektur getrennten Speicher für Programme und Daten.

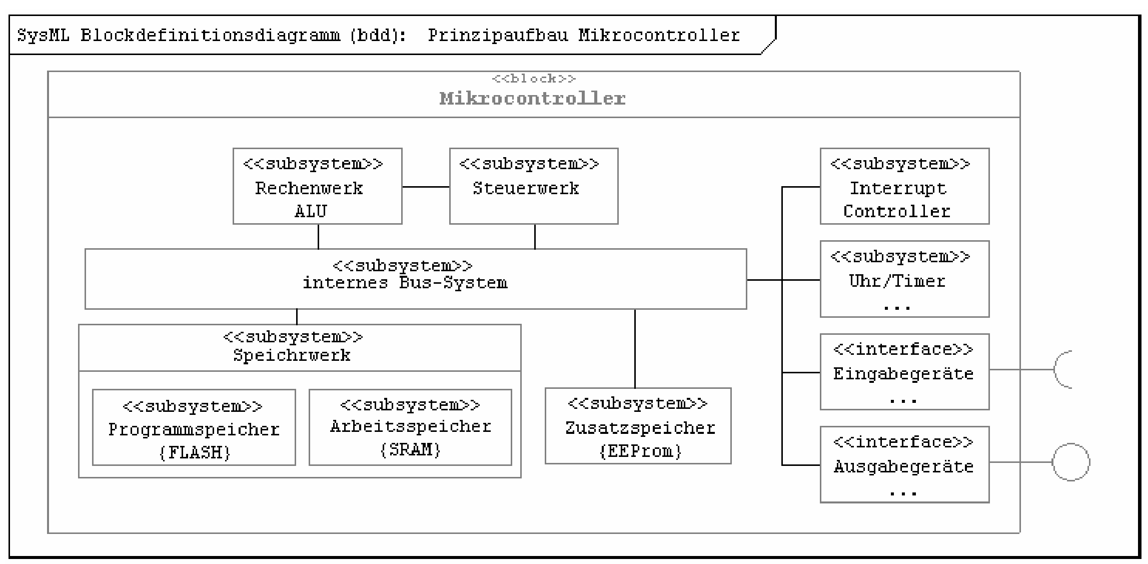


Abbildung: Prinzipaufbau eines Mikrocontrollers

Um das Sprungparadigma zu verstehen ist es nötig, sich mit dem Arbeitszyklus eines Digitalrechners auseinanderzusetzen. Nachdem ein Digitalrechner mit Spannung versorgt oder ein Reset ausgeführt wurde, befinden sich die für die Steuerung der Zentraleinheit entschiedenen Register im Ursprungszustand:

- SP (Stackpointer) = 0
- PC (Program Counter) = 0
- IR (Instruction Register) = 0
- Status Register (SREG) = 0
- Register Set [r0..r31] = {0}

Die Zentraleinheit führt, solange Spannung anliegt und kein Resetsignal vorliegt, ununterbrochen den abgebildeten Arbeitszyklus aus. Dieser Arbeitszyklus führt dazu, dass ab der Adresse 0 die im Programmspeicher befindlichen Befehle nacheinander abgearbeitet werden. Dabei wird der Befehlszähler (Program Counter) bei jedem Zyklus, ohne zutun des Programmierers, automatisch um eins erhöht und bringt die Zentraleinheit auf den nächsten auszuführenden Befehl. Dabei werden die Befehle tatsächlich in der Reihenfolge ihrer Speicherung im Programmspeicher abgearbeitet (sequenzielle Verarbeitung). Mit bestimmten Befehlen kann jedoch der Inhalt des Befehlszählers (Program Counter) direkt verändert werden. Diese speziellen Befehle bewirken eine sprungartige Neuausrichtung der Zentraleinheit auf einen Befehl, der an einer beliebigen Position im Programmspeicher liegen kann. Ein Sprung zu einer kleineren Adresse im Programmspeicher bewirkt, dass zu Befehlen zurückgekehrt wird, die bereits abgearbeitet wurden. Diese Befehle werden somit wiederholt. Ein Sprung zu einer größeren Adresse im Programmspeicher bewirkt, dass die Befehle bis dorthin nicht ausgeführt, also ausgelassen werden. Die möglichen Maschinenbefehle (Instruction Set) einer Zentraleinheit werden bei der Programmierung als sogenannte Assemblerbefehle geschrieben oder generiert.

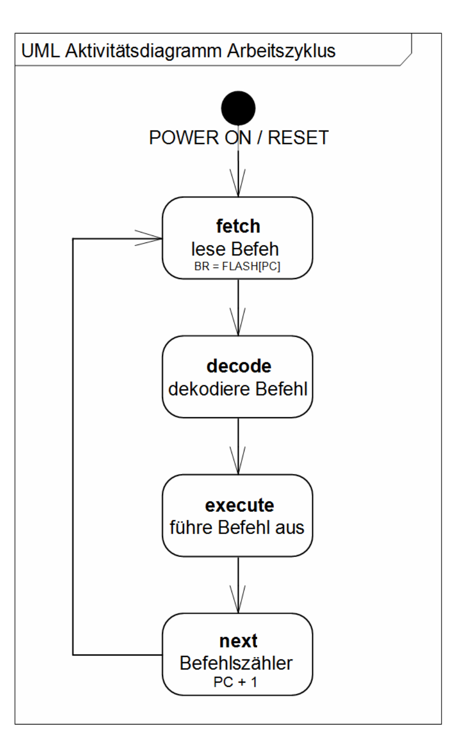


Abbildung: Arbeitszyklus einer Zentraleinheit

.....

4.2 Grundzüge Strukturierter Programmiersprachen

Die typischen Merkmale einer Strukturierten Programmiersprache sollen anhand der in der Mikrocontrollerprogrammierung weit verbreiteten Sprache C aufgezeigt werden.

Was ist C?

C ist eine höhere Programmiersprache mit breiten Einsatzmöglichkeiten. Die grundlegende Komponente (Modulkonzept) dieser Programmiersprache sind Funktionen. Diese kann man als Befehle mit Parametern aufrufen. Die Sprache selbst abstrahiert von der konkreten Maschinensprache und ist dadurch portabel (Übertragbarkeit des C-Codes auf verschiedenen Rechnertypen). Auf Sprünge wird zur Bildung von Algorithmen verzichtet (strukturierte Programmierung GOTO-freie Programmierung).

Einsatzgebiete

C eignet sich besonders als Sprache für die Betriebssystementwicklungen (dafür wurde sie ursprünglich entwickelt). Weiterhin lassen sich numerische Verfahren, Textverarbeitung und Datenbanken aber auch Spiele effizient in C realisieren. C verlangt jedoch vom Programmierer eine gewisse Disziplin zum Beispiel im Umgang mit Speicherressourcen. In anderen Sprachen (BASIC, JAVA, C#) wird das Speichermanagement dem Programmierer abgenommen. Dafür ist der Speicherbedarf solcher Programme auch oft um ein vielfaches höher als bei C und die Programme selbst laufen viel langsamer als ein vergleichbares C-Programm.

Editor, Quellcode, Kompiler, Linker, Programm

Die Schritte von der Idee zum lauffähigen Programm sind in der Regel Folgende:

	Arbeitsschritt	**Werkzeug**
-	Aufgabe, Problem verstehen	Kopf ;-)
-	Lösungsidee entwerfen	Stift, Tafel oder CASE-Tool
-	Quellcode schreiben	Editor / Progarmmierumgebung
-	Quellcodemodule übersetzen	Compiler
-	Module zu Programm binden	Linker
-	Fehler im Programm suchen	Debugger, Simulator
-	Programm zum Controller übertragen	Programmer,

Zuerst muss der Algorithmus entworfen werden. Ist dieser klar, kann der Quellcode erstellt werden. Das geschieht mit einem Quelltexteditor. Danach wird ein Compiler und Linker aufgerufen. Der Kompiler überprüft den Code auf Syntaxfehler und übersetzt diesen in eine Objekt-Datei (Maschinencode). Dieser Objektcode wird vom Linker weiterverarbeitet. Durch den Linker wird der Objektcode mit den nötigen Bibliotheksfunktionen verbunden, um somit ein ausführbares Programm zu erzeugen. Bei der Programmierung in AVR C liegen im Projektverzeichnis Dateien mit folgenden Endungen vor:

`*.c, *.cc, *.cpp`	Quelltext (Programm)
`*.h, *.hh`	Header-Dateien (Definitionen)
`*.o, *.obj`	Objekt-Dateien (Maschinencode-Modul)
`*.a, *.lib`	Bibliothek (Sammlung von Objektdateien)
`*.hex, *.bin, *.elf`	ausführbares Mikrocontrollerprogramm

C Sprachumfang

Die Sprache C selbst umfasst, wie die meisten strukturiereten Hochsprachen, eine überschaubare Menge an Befehlen und Schlüsselworten. Diese Bezeichner haben eine bestimmte Funktion oder Bedeutung und können vom Programmierer nur in den vorgesehen Bedeutungen verwendet werden. Eine Redeklarion oder Neuverwendung der Bezeichner die zum Sprachumfang der Programmiersprache gehören ist ausgeschlossen, deshalb nennt man diese Bezeichner auch reservierte Worte. Im Folgenden ist die Sprache C fast komplette aufgezeigt:

Erlaubte Zeichen:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z

Erlaubte Sonderzeichen:

! " % & / ( ) [ ] { } \ ? = ' # + * ~ – _ . : ; , | < > ^

Dezimalziffern:

1 2 3 4 5 6 7 8 9 0

Schlüsselworte und Bezeichner (für den Anfang wichtige fett):

auto	double	**int**	struct
break	**else**	long	**switch**
case	enum	register	typdef
char	extern	**return**	union
const	float	short	**unsigned**
continue	**for**	signed	**void**
default	goto	sizeof	**volatile**
do	**if**	static	**while**

Binäre arithmetische Operatoren (zwei Operanten nötig)

+	Addition,	**-**	Subtraktion,
*****	Multiplikation,	**/**	Division,
%	Modulo		

Unäre arithmetische Operatoren (nur ein Operant nötig)

+,-	Vorzeichen,	**++**	Inkrement,
--	Dekrement		

Bit-Operatoren

&	and,	**|**	or,
~	not,	**^**	xor,
>>	shift right,	**<<**	shift left

Logische Operatoren

==	ist gleich	**>**	größer als
<	kleiner als	**>=**	größer gleich
<=	kleiner gleich	**!=**	ungleich

Wertzuweisungen

=

Trennzeichen

;	Befehlsende
{	Blockanfang Geltungsbereich
}	Blockende Geltungsbereich
(	Anfang Parameterliste
)	Ende Parameterliste
,	Trennzeichen Parameter

Zeichenketten
“Hallo“ konstanter String, dieser wird in doppelte Hochkomma gestellt
’A’ konstantes Zeichen, dies wird in einfache Hochkomma gestellt

Zahlen
123 Integerzahl Dezimaldarstellung
1.23 Gleitkommazahl
0xA0 Integerzahl Hexadezimaldarstellung
0b01010110 Integerzahl Binärdarstellung

Kommentare
// bis Zeilenende
/* Komentaranfang
***/** Kommentarende

Compileranweisungen (Auswahl)
#include, #define, #ifndef, #ifdef, #else, #endif, …

Alle darüber hinausgehenden Schlüsselworte wie **PORTB** oder **sprintf(…)** sind Definitionen oder Funktionen die zwar in C geschrieben, aber nicht zum Umfang der Sprache selbst gehören. Manchmal sind diese spezifisch für ein Zielsystem, eine Zielplattform oder ähnliches. Die zusätzlichen Funktionen stehen als Bibliotheken zur Verfügung. Bibliotheken sind nicht immer zwingend standardisiert und können von Anbieter zu Anbieter gravierende Unterschiede aufweisen. Zusätzlich legt der Programmierer eigene Bezeichner für Variablen und selbstdefinierte Funktionen und Typen fest.

Anweisungen und Ausdrücke
Eine Anweisung ist die kleinste Programmiereinheit. Mehrere Anweisungen können mit geschweiften Klammern in Blöcke zusammengefasst werden. Jede Anweisungen muss immer durch ein Semikolon abgeschlossen werden. Ein Ausdruck ist eine der Sprachregeln folgende Aneinanderreihung von Bezeichnern, Zahlen, Strings sowie Operatoren. Ausdrücke liefern immer ein Ergebnis.

eine Anweisung: **`zahl = 5 + 8;`**
ein Ausdruck: **`5 + 8`**

Blockbildung
Da die Sprache C dem Modulkonzept folgt, werden bei der Programmentwicklung konsequent Module als Blöcke gebildet. Dabei ist das Mindeste die Festlegung eines Anfangs und Endes für den Modulblock. Das geschieht mit den geschweiften Klammern. Soll ein funktionale Einheit (Modul) nicht nur einmal (inline) sondern mehrfach verwendet werden, bekommt diese zusätzlich einen Namen und Parameter.

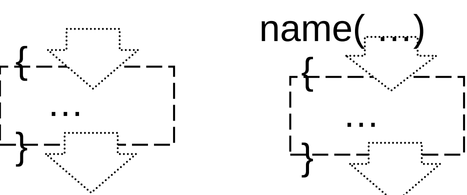


Abbildung: Modulbildung in C

Die wichtigsten Datentypen in AVR C

char Datentyp für eine Textvariable (Character, Buchstabe).
8 Bit,
Wertebereich: -128...127
Beispiel: char buchstabe = 'K';

int Datentyp für eine Ganzzahl (Integer).
16 Bit,
Wertebereich: -32768...+32767
Beispiel: int alter = 37;

float Datentyp für eine Fließkommazahl, die Kommastelle symbolisiert ein Punkt
32 Bit,
Wertebereich: $3.4*10^{-38}...3.4*10^{38}$
Beispiel: float alter = 37.5;

Eine ausführlichere Auflistung von Datentypen steht Ihnen im Anhang zur Verfügung. Häufig sind bei der Deklaration von Variablen noch die Schlüsselworte *unsigned* und *volatile* anzutreffen.

unsigned (vorzeichenlos) legt fest, dass ein Wert kein Vorzeichen besitzt also nur positive Zahlen darstellen kann. Der Typ unsigned selbst repräsentiert ein Bit.

Beispiele:

```
char   wert1;          // 8 Bit, Wertebereich: -128...127
unsigned char wert2;  // 8 Bit, Wertebereich: 0...255
int   wert3;           // 16 Bit, Wertebereich: -32768...+32767
unsigned int wert4;   // 16 Bit, Wertebereich: 0...+32767
```

volatile (flüchtig) legt fest, dass der Wert einer Variablen durch andere Quellen oder Ereignisse außerhalb der Funktion verändertwerden kann. Die Wirkung der Deklaration als flüchtig, d.h. das bei jeder Veränderung der Wert sofort in seinen zugehörigen Speicher im SRAM zurückgeschrieben wird und vor jeder Benutzung von dort gelesen. Desweiteren sind diese Variablen von jeglicher Optimierung des Compilers ausgeschlossen.

Beispiel:

```
volatile int  wert5; // 16 Bit, von Optimierung ausgeschlossen
```

bool Wahrheitswert, 8 Bit, Wertebereich: `true | false`

Da bei C die Übersetzung von bestimmten Datentypen (z.B. int) von der Verarbeitungsbreite der Zielplattform (8, 16, 32, 64 Bit) abhängig ist, gibt es für die Wahrung der Portierbarkeit von C-Codes in AVR C spezielle Typdeklarationen, welche die Speichergröße plattformunabhängig festlegen. Hier einige Beispiele:

int8_t Integerzahl, 8 Bit, Wertebereich: -128 ... +127
uint8_t Integerzahl, 8 Bit, Wertebereich: 0 ... +255
int16_t Integerzahl, 16 Bit, Wertebereich: -32768 ... +32767
uint16_t Integerzahl, 16 Bit, Wertebereich: 0 ... +65535

Alle weiteren Kenntnisse sollten Sie anhand der einzelnen Beispielen erlernen können ;-)

Ein einfaches C Programm für AVR Mikrocontroller:

```
1    /* Grundstruktur eines AVR C Programms */
2    #include <avr/io.h>
3    main ()
4    {
5         // Initalisierung PORTB = Ausgang
6         DDRB = 0xFF;
7         while (1==1)
8         {
9            PORTB = 0xFF; // Ausgabe
10        }
11   }
```

Erläuterung der verwendeten Befehle:

Zeile1: **/*Grundstruktur eines AVR C Programms */**
Mit /* fügt man Kommentare in den Quelltext ein, mit */ wird der Kommentar wieder beendet. Mit // kann man einen Kommentar einfügen, der nur über eine Zeile geht. Ein Kommentar dient zur Dokumentation des Programmes und wird vom Compiler nicht in Maschinencode übersetzt.

Zeile 2: **#include <avr/io.h>**
Dies ist eine sogenannte Compiler-Direktive. #include weist den Compiler an, eine vorhandene Datei hier "einzufügen". *.h Dateien sind Header-Dateien, sie enthalten Bibliotheks-Funktionen und vordefinierten Deklarationen wie zum Beispiel Registernamen. Die Headerdatei <io.h> verweist auf alle wichtigen Funktionen und Definitionen zur Ein- und Ausgabe über Ports und Register des ausgewählten AVR-Controllers.

Zeile 3: **main()**
Dies ist die Deklaration der Hauptfunktion (Hauptprogramm). Sie wird nach dem Start des Systems immer als erstes abgearbeitet. Ein Beenden dieser Funktion ist bei Controlleranwendungen ohne Betriebssystem nicht sinnvoll. Daraus ergibt sich die Notwendigkeit einer Unendlichschleife in der die geforderte Verarbeitungsaufgabe ständig ausgeführt wird. Das ist letztlich das „Betriebssystem".

Zeile 4: **{**
Mit der geschweiften Klammer beginnt der Block/Körper einer Funktion. In ihm werden alle Anweisungen festgelegt, die die Funktion nacheinander abzuarbeiten hat. Beachten Sie, dass zu jeder öffnenden Klammer auch eine schließende Klammer gehört. Alles was sich innerhalb der öffnenden und schließenden Klammer befindet gehört zu dem Block. Blöcke können wiederum Blöcke enthalten.

Zeile 5: **// Initialisierung;**
Bei dieser Zeile handelt es sich wiederum um einen Kommentar. Der Unterschied zum ersten Kommentar ist, dass hier ein anders Symbol verwendet wurde. Man benötigt hier kein Endsymbol da dieses Kommentarsymbol automatisch bis Zeilenende gilt.

Zeile 6: **DDRB = 0xFF;**
Die Zeile 6 enthält eine Anweisung die dem Bezeichner DDRB den Wert 0xFF zuweist. Zu beachten ist das die Anweisung (Befehl) mit einem Semikolon abzuschließen ist.

Zeile 11: **while (1==1)**
Das Schlüsselwort while zeigt eine Wiederholbedingung in C an. Die eigentliche Bedingung wie lange bzw. wie oft die Anweisung im Schleifenblock wiederholt werden soll folgt in runden Klammern. Die Bedingung 1==1 ist immer wahr. Zu Deutsch, wiederhole solange 1 gleich 1 ist. Also für Immer ;-) Es ist möglich hier auch nur eine 1 oder das Schlüsselwort **true** zu verwenden da das Ergebnis dieser Ausdrücke immer dasselbe ist.

Zeile 8: **{**
Die geschweifte Klammer öffnet einen Block, in dem Fall den Schleifenblock für die Wiederholung while(1==1).

Zeile 9: **PORTB = 0xFF; // Ausgabe**
Die Zeile 9 enthält den fortlaufend auszuführenden Befehl in der Schleife. Der Schleifenblock kann mehr als einen Befehl enthalten. Weitere Befehle können innerhalb der Klammer eingefügt werden und müssen einzeln mit einem Semikolon abgeschlossen sein. Die Zeile enthält einen zusätzlichen Kommentar. Es ist deutlich zu sehen, dass der erste Teil der Zeile eine gültige Anweisung ist und ab dem Kommentarsymbol eine Erläuterung folgt.

Zeile 10: **}**
Diese geschweifte Klammer schließt den Schleifenblock. Alles was zwischen der öffnenden Klammer und der dazugehörigen schließenden Klammer steht wird so lange wiederholt bis die Bedingung der Schleife eine Null (false) ergibt. Im Fall der Hauptschleife läuft diese „unendlich".

Zeile 11: **}**
Die letzte geschweifte Klammer bedeutet das Ende des Funktionskörpers. Alles was nach dieser Klammer steht gehört nicht mehr zur Funktion main.

Whitespacezeichen und Klammern

In C werden sogenannte Whitespace-Zeichen (Leerzeichen, Tabulatoren und Enter) einfach ignoriert. Sie dienen ausschließlich der Quellcodeformatierung und haben keinen Einfluss auf den Programmablauf. Es ist unerheblich, wo und ob Sie Leerzeichen oder neue Zeilen einfügen. Innerhalb von Schlüsselwörtern, Zeichenketten oder Variablennamen dürfen Sie allerdings keine Leerzeichen, Tabulatoren oder Enter verwenden. Möglich wäre für unser Mini-Programm also auch folgende Schreibweise:

```
#include <avr/io.h>
main(){DDRB=0xFF;while(true){PORTB=0xFF;}}
```

Diese Schreibweise ist zwar zulässig aber wenig übersichtlich.

.....

5.2 Grundzüge objektorientierter Programmiersprachen

Ausgangspunkt für das Verstehen einer objektorientierten Programmiersprache ist immer das objektorientierte Paradigma. Das Basiskonzept stellt sozusagen das SOLL und die konkrete Sprache das IST dar. Gehen Sie davon aus, dass eigentlich in keiner derzeit verfügbaren objektorientierten Sprache auch alle in der Theorie formulierten Konzepte bereits komplett umgesetzt sind. Man sollte sich auf jeden Fall davor hüten von den Möglichkeiten und den Einschränkungen einer konkreten Sprache auf das Konzept zu schließen. Wesentliche Aspekte objektorientiertes Sprachen sollen im Folgenden anhand der Sprache C++ aufgezeigt werden. Für das Verständnis von C++ ist weiterhin wichtig zu wissen, dass C++ die Sprache C beinhaltet. C++ ist die objektorientierte Erweiterung der Sprache C.

Zusätzlicher Sprachumfang von C++ (Auswahl, wichtige fett)

bool	**true**	**false**			
class	**new**	**delete**	enum	template	**this**
virtual	operator	**private**	**protected**	**public**	
namespace	using	catch	throw	try	

Deklarieren von Klassen in C++

Eine Person hat einen Namen und Voramen sowie ein Geburtsjahr. Sie kann nach ihrem Namen und dem Geburtsjahr gefragt werden.

```
// Klasse Name { Bauplan: Struktur, Verhalten };
class Person
{
	// Attribute: Sichtbarkeit, Typ, Name
	protected: char nachName[20];
	protected: char vorName[20];
	protected: int geburtsJahr;
	// Konstruktor, Destruktor
	Person ()
	{
		// hier Initialisierungen durchführen
	}
	~Person ()
	{
		// hier Deinitialisierungen durchführen
	}
	// Operationen: Sichtbarkeit, Typ, Name, Parameter
	public: char* getNachName()
	{
		return nachName;
	}
	public: char* getVorName()
	{
		return vorName;
	}
	public: int getGeburtsJahr()
	{
		return geburtsJahr;
	}
};
```

Vererbung in C++

Ein Mitarbeiter ist eine Person. Zusätzlich hat er eine Mitarbeiternummer. Diese kann ihm zugewiesen werden und er kann nach seiner Miarbeiternummer gefragt werden.

```
// Klasse Name : Basisklasse { Bauplanerweiterung };
class Mitarbeiter : public Person
{
	// Attribute: Sichtbarkeit, Typ, Name
	protected: int maNummer;
	// Operationen: Sichtbarkeit, Typ, Name, Parameter
	public: void setMaNummer( int nummer )
	{
		maNummer = nummer;
	}
	public: int getMaNummer()
	{
		return maNummer;
	}
};
```

Definieren von Objekte C++

Meier vom Typ Mitarbeiter; Die Instanz (Variable) *meier* ist das Objekt. Der Typ der Instanz (Variable) ist der Klassenname *Mitarbeiter*. Die Instanz *meier* hat alle Merkmale einer Person und eines Mitarbeiters.

```
// ...
Mitarbeiter meier;
// ...
```

Aggregationen und Kapselung in C++

Mitarbeiter haben ein Konto. Der Kontostand ist Privatsache. Einzahlen darf jeder.

```
// Klasse Name : Basisklasse { Bauplanerweiterung };
class Konto
{
	private: float kontostand;
	public: void einzahlen( float betrag )
	{
		kontostand += betrag;
	}
};
// -----------------------------------------------
class Mitarbeiter : public Person
{
	// ...
	// Aggregation Konto
	public: Konto konto;
	// ...
};
```

Assoziationen in C++

Mitarbeiter kennen den Betriebsrat. Das * gibt an, dass es sich nur um einen Zeiger (die Adresse im Speicher) auf eine Instanz handelt. Die Instanz vom Konto ist in der Klasse Mitarbeiter vollständig enthalten. Wird ein Mitarbeiter angelegt, hat er auch automatisch ein Konto. Der Betriebsrat existiert außerhalb des Mitarbeiters. Der Mitarbeiter muss sich die Adresse des Betriebsrates erst noch beschaffen, um mit diesem Nachrichten auszutauschen. Das Konto kann sofort benutzt werden.

```
class Mitarbeiter : public Person
{
    // ...
    // Aggregation (hat) Konto
    public: Konto konto;
    // im Verleich dazu eine Assoziation (kennt)
    public: Betriebsrat* adresseBetriebsrat;
    // ...
};
```

Nachrichten in C++

Meier bekommt eine Prämie und spendet davon etwas an den Betriebsrat.
Beachten Sie die unterschiedlichen Zugriffssymbole.

```
// ...
Mitarbeiter meier;
// Nachricht 500 Euro einzahlen an das Konto von Meier
meier.konto.einzahlen(500);
// Nachricht 5 Euro spenden an die Adresse des Betriebsrates
adresseBetriebsrat->spenden(5);
// ...
```

Ein einfaches C++ Programm für AVR Mikrocontroller:

```
01 /* Grundstruktur eines AVR C++ Programms */
02 #define F_CPU 16000000
03 #include <avr/io.h>
04 class Controller
05 {
06     public: void powerOn()
07     {
08         // Initalisierung B.0 = Ausgang
09         ddrB.bit0 = 1;
10         run();
11     }
12     //-----------------------------------
13     public: void run()
14     {
15         while (true)
16         {
17             portB.bit0 = 1; // Ausgabe
18         }
19     }
20 };
21 //----------------------------------------
22 Controller app;
23 main ()
24 {
25     app.powerOn();
26 }
```

Erläuterung der verwendeten Befehle:

Zeile1: **/*Grundstruktur eines AVR C++ Programms */**
Mit /* fügt man Kommentare in den C++ Quelltext ein, mit */ wird der Kommentar wieder beendet. Mit // kann man einen Kommentar einfügen, der nur über eine Zeile geht. Ein Kommentar dient zur Dokumentation des Programms und wird vom Compiler nicht in Maschinencode übersetzt.

Zeile 2: **#define F_CPU 16000000**
Dies ist eine sogenannte Compiler-Direktive. *#define* weist den Compiler an, einem symbolischen Namen einen Wert zuzuordnen. Im Quelltext kann dann der Name statt des Wertes verwendet werden. Es handelt sich um eine Konstante.

Zeile 3: **#include <avr/io.h>**
Dies ist ebenfalls eine Compiler-Direktive. *#include* weist den Compiler an, eine vorhandene Datei hier "einzufügen". **.h* Dateien sind Header-Dateien, sie enthalten Bibliotheks-Funktionen und vordefinierten Deklarationen wie zum Beispiel Registernamen. Die Headerdatei *io.h* verweist auf alle wichtigen Funktionen und Definitionen zur Ein- und Ausgabe über Ports und Register des ausgewählten AVR-Controllers.

Zeile 4: **class Controller**
Das Schlüsselwort *class* leitet eine Klassendeklaration ein. Diese ist gefolgt vom Klassennamen und dem Klassenkörper in geschweiften Klammern. Beachten Sie, dass Klassendeklarationen immer mit einem Semikolon abgeschlossen werden müssen.

Zeile 5: **{**
Mit der geschweiften Klammer beginnt der Block/Körper der Klasse. In ihm werden Struktur und Verhalten der Klasse deklariert. Beachten Sie, dass zu jeder öffnenden Klammer auch eine schließende Klammer gehört. Alles was sich innerhalb der öffnenden und schließenden Klammer befindet, gehört zu dem Block. Blöcke können wiederum Blöcke enthalten.

Zeile 6: **public: void powerOn()**
Die Deklaration einer Mitgliedsfunktion (Operation/Methode) einer Klasse muss in deren Klassenkörper erfolgen. Es ist mindestens die Sichtbarkeit, Typ und Name der Operation anzugeben. Parameter werden in runden Klammern mit Typ und Name aufgelistet. Wenn die Operation keine Parameter besitzt, müssen leere Klammern notiert werden. Diese Mitgliedfunktion soll aufgerufen werden wenn der Controller eingeschaltet wurde.

Zeile 7: **{**
Mit der geschweiften Klammer beginnt der Körper dieser Funktion. In ihm werden alle Anweisungen festgelegt, die die Funktion nacheinander abzuarbeiten hat. Beachten Sie, dass zu jeder öffnenden Klammer auch eine schließende Klammer gehört. Alles was sich innerhalb der öffnenden und schließenden Klammer befindet, gehört zu dem Block. Blöcke können wiederum Blöcke enthalten.

Zeile 8: **// Initalisierung B.0 = Ausgang**
Bei dieser Zeile handelt es sich wiederum um einen Kommentar. Der Unterschied zum ersten Kommentar ist, dass hier ein anders Symbol verwendet wurde. Man benötigt hier kein Endsymbol, da dieses Kommentarsymbol automatisch bis Zeilenende gilt.

Zeile 9: **ddrB.bit0 = 1;**
Dem *bit0* der Instanz *ddrB* wurde eine 1 zugewiesen. Das Objektorientierte Paradigma geht davon aus, dass ein System aus strukturierten Instanzen besteht. Jede Instanz aggrigiert die Elemente, für die sie Verantwortlich ist.

Zeile 10: **run();**
Aufruf der Mitgliedsfunktion *run*. Es handelt sich hier um eine Nachricht an sich selbst. Alternativ kann per Zeiger auf sich selbst verwiesen werden. Daraus ergibt sich die ebenfalls mögliche Schreibweise *this->run();*

Zeile 11: **}**
Diese geschweifte Klammer bedeutet das Ende des Funktionskörpers. Alles was nach dieser Klammer steht gehört nicht mehr zur Funktion.

Zeile 12: **//----------------------------------**
Dieser Kommentar dient der Übersichtlichkeit. Er trennt die Operationen optisch voneinander.

Zeile 13: **public: void run()**
Die Operation run kann aufgerufen werden nach dem alle Initialisierungen erledigt sind. Der Controller arbeitet dann in der Hauptschleife bis zum Abschalten der Spannungsversorgung. Ein Beenden dieser Funktion ist bei Controlleranwendungen ohne Betriebssystem nicht sinnvoll. Daraus ergibt sich die Notwendigkeit einer Unendlichschleife in der die geforderte Verarbeitungsaufgabe ständig ausgeführt wird. Das ist letztlich das „Betriebssystem“ (vgl. Polling). Wenn die Funktionalität der Controlleranwendung ereignisorientiert ist (vgl. Interrups), dann kann die Hauptschleife auch leer sein darf jedoch trotzdem nicht weg gelassen werden (vgl. Leerlaufprozess in Betriebssystemen).

Zeile 14: **{**
Das ist öffnende Klammer für den Funktionskörper der Operation *run*.

Zeile 15: **while (true)**
Das Schlüsselwort while zeigt eine Wiederholbedingung in C an. Die eigentliche Bedingung wie lange bzw. wie oft die Anweisung im Schleifenblock wiederholt werden soll, folgt in runden Klammern. Die Bedingung *true* ist immer wahr. Zu Deutsch, wiederhole solange wahr gleich wahr ist. Also für Immer ;-) Es ist möglich hier auch nur eine 1 oder das den Ausdruck 1 == 1 zu verwenden, da das logische Ergebnis dieser Ausdrücke immer dasselbe ist.

Zeile 16: **{**
Die geschweifte Klammer öffnet einen Block, in dem Fall den Schleifenblock für die Wiederholung *while(true)*.

Zeile 17: **portB.bit0 = 1; // Ausgabe**
Die Zeile 17 enthält den fortlaufend auszuführenden Befehl in der Schleife. Der Schleifenblock kann mehr als einen Befehl enthalten. Weitere Befehle können innerhalb der Klammer eingefügt werden und müssen einzeln mit einem Semikolon abgeschlossen sein. Die Zeile enthält einen zusätzlichen Kommentar. Es ist deutlich zu sehen, dass der erste Teil der Zeile eine gültige Anweisung ist und ab dem Kommentarsymbol eine Erläuterung folgt.

Zeile 18: **}**
Diese geschweifte Klammer schließt den Schleifenblock. Alles was zwischen der öffnenden Klammer und der dazugehörigen schließenden Klammer steht wird so lange wiederholt, bis die Bedingung der Schleife eine Null (false) ergibt. Im Fall der Hauptschleife läuft diese „unendlich".

Zeile 19: **}**
Diese geschweifte Klammer bedeutet das Ende des Funktionskörpers. Alles was nach dieser Klammer steht gehört nicht mehr zur Funktion.

Zeile 20: **};**
Die letzte geschweifte Klammer bedeutet das Ende des Klassenkörpers. Alles was nach dieser Klammer steht gehört nicht mehr zur Klasse.

Zeile 21: **//-----------------------------------**
Dieser Kommentar dient der Übersichtlichkeit.

Zeile 22: **Controller app;**
In Zeile 22 wurde eine globale Instanz vom Typ *Controller* angelegt. Ab hier kann von über all auf die Applikation *app* zugegriffen werden.

Zeile 23: **main ()**
Dies ist die Deklaration der strukturierten Hauptfunktion (Hauptprogramm). Sie wird nach dem Start des Systems immer als erstes abgearbeitet. Eine klassische Hauptfunktion ist nach wie vor nötig. Diese ist dafür verantwortlich die aktive Klasse zu initialisieren und zu starten.

Zeile 24: **{**
Mit dieser geschweiften Klammer beginnt der Körper der Funktion *main*. In ihm werden alle Anweisungen festgelegt, die die Funktion nacheinander abzuarbeiten hat. Beachten Sie, dass zu jeder öffnenden Klammer auch eine schließende Klammer gehört. Alles was sich innerhalb der öffnenden und schließenden Klammer befindet gehört, zu dem Block. Blöcke können wiederum Blöcke enthalten.

Zeile 25: **app.powerOn();**
Es wird an die globale Instanz *app* die Nachricht *powerOn* gesendet.

Zeile 26: **}**
Die letzte geschweifte Klammer bedeutet das Ende des Funktionskörpers. Alles was nach dieser Klammer steht gehört, nicht mehr zur Funktion.

5.3 Einführung in die UML

Die Unified Modeling Language ist ein Satz von Darstellungsregeln (Notation) zur Beschreibung objektorientierter Softwaresysteme. Ihre ursprünglichen Autoren Grady Booch, James Rumbaugh, Ivar Jacobson verfolgten mit der eigens gegründeten Firma Rational anfangs vor allem kommerzielle Ziele. Sie übergaben die UML jedoch im weiteren Verlauf der Entwicklung als offenen Standard an eine nicht kommerzielle Organisation, der Object Management Group (www.omg.org). Im Jahre 1996 wurde die UML durch die OMG zu einem internationalen Standard erhoben. Die OMG entwickelt die UML und auf der UML basierende Konzepte und Standards weiter. Die UML soll nach den Wünschen der Autoren eine Reihe von Aufgaben und Zielen verfolgen, so zum Beispiel:

- Bereitstellung einer universellen Beschreibungssprache für alle Arten objektorientierter Softwaresysteme und damit eine Standardisierung,
- Vereinigung der beliebtesten Darstellungstechniken (best practice),
- ein für zukünftige Anforderungen offenes Konzept
- Architekturzentrierter Entwurf

Durch die UML sollen Softwaresysteme besser

- analysiert
- entworfen und
- dokumentiert werden

die Unified Modeling Language …

- ist NICHT perfekt, wird aber immer besser
- ist NICHT vollständig, wird aber immer umfangreicher
- ist KEINE Programmiersprache, man kann mit ihr aber programmieren
- ist KEIN vollständiger Ersatz für eine Textbeschreibungen, man kann mit ihr aber immer mehr beschreiben
- ist KEINE Methode oder Vorgehensmodell, mit ihr wird die Systementwicklung aber viel methodischer
- ist NICHT für alle Aufgabenklassen geeignet, sie dringt jedoch in immer mehr Aufgabengebiete vor

Die UML spezifiziert selbst keine explizite Diagrammhierarchie. Die Diagramme der UML werden verschiedenen semantischen Bereichen zugeordnet

- **Structure (Strukturdiagramme)**
    - **Class Diagram** (Klassendiagramm)
      wichtigstes Diagramm: Klassen und ihre Beziehungen untereinander)
    - **Package Diagram** (Paketdiagramm)
      gliedert Softwaresysteme in Untereinheiten
    - **Object Diagram** (Objektdiagramm)
      Objekte, Assoziationen und Attributwerte zu einem bestimmten Zeitpunkt während Laufzeit
    - **Composite Structure Diagram** (Kompositionsstrukturdiagramm)
      Abbildung innerer Zusammenhänge einer komplexen Systemarchitektur, Darstellung von Design Patterns
    - **Component Diagram** (Komponentendiagramm)
      Komponenten und ihre Beziehungen und Schnittstellen
    - **Deployment Diagram** (Verteilungsdiagramm)
      Einsatzdiagramm, Knotendiagramm, Laufzeitumfeld

- **Behavior (Verhaltensdiagramme)**
  - **Use Case Diagram** (Use-Case-Diagramm, Anwendungsfalldiagramm)
    stellt Beziehungen zwischen Akteuren und Anwendungsfällen dar
  - **Activity Diagram** (Aktivitätsdiagramm)
    beschreibt Ablaufmöglichkeiten, die aus einzelnen Aktivitäten/Schritten bestehen
  - **State Machine** (Zustandsdiagramm, Zustandsautomat)
    zeigt eine Folge von Zuständen eines Objekts
  - **Sequence Diagram** (Sequenzdiagramm)
    wichtigstes Interaktionsdiagramm: zeigt den zeitlichen Ablauf von Nachrichten zwischen Objekten
  - **Communication Diagram** (Kommunikationsdiagramm, früher Kollaborationsdiagramm)
    Interaktionsdiagramm: zeigt Beziehungen und Interaktionen zwischen Objekten
  - **Timing Diagram** (Timingdiagramm, Zeitverlaufsdiagramm)
    Interaktionsdiagramm mit Zeitverlaufskurven von Zuständen
  - **Interaction Overview** (Interaktionsübersichtsdiagramm)
    Interaktionsdiagramm zur Übersicht über Abfolgen von Interaktionen, ähnlich Aktivitätsdiagramm

Die Ableitung weiterer Diagrammarten ist durchaus möglich bzw. sogar gewollt und wird von der UML auch entsprechend unterstützt. Dazu stellt die UML Möglichkeiten zur Anpassung des Modellgerüstes (Metamodell) zur Verfügung. Das wichtigste Werkzeug zur Anpassung der UML an spezifische Anforderungen (Profiling) ist das Stereotypkonzept. Dabei entstehen so genannte UML-Profile.

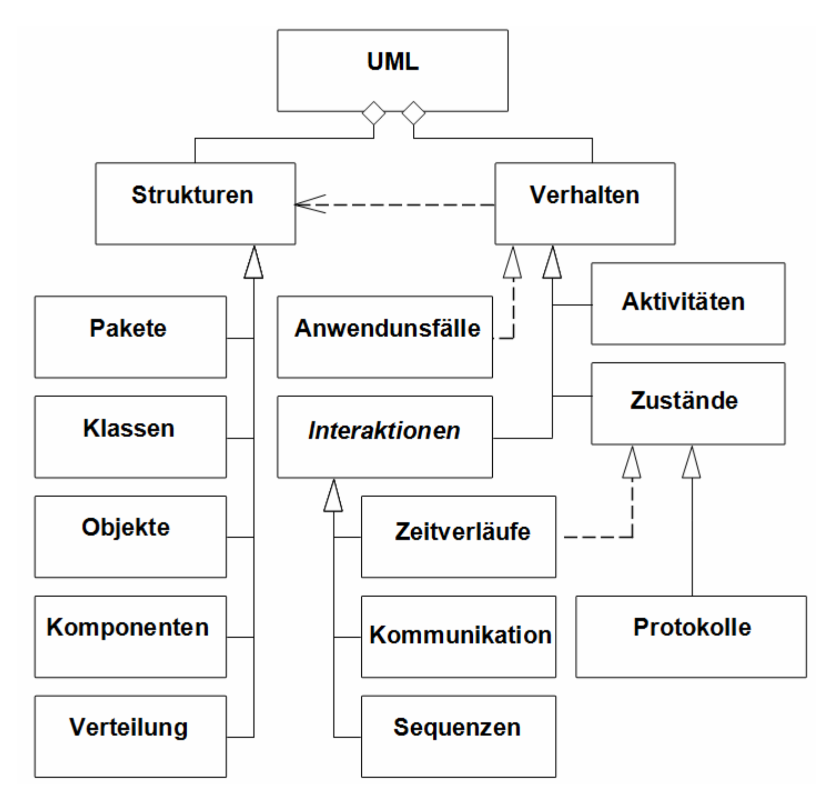


Abbildung: Taxonomie der UML

.....

5.5 Modellierung mit dem Zustandsdiagramm der UML

Alles um uns herum und auch wir selbst befindet sich in einer Menge bestimmter Zustände (engl. state). Sie als Leser sind in diesem Moment wach und hoffentlich konzentriert. Das Regenwasser an der Fensterscheibe vor Ihrem Schreibtisch ist im Zustand flüssig usw. Jeder dieser Zustände ist durch bestimmte Merkmale (engl. feature) gekennzeichnet. Das sind zum einen Strukturmerkmale, also im Sinne der Objektorientierung Attribute, aber auch Verhaltensmerkmale die wir an dieser Stelle als Aktivitäten bezeichnen wollen. Diese Merkmale kennzeichnen den Zustand. Zum Beispiel ist das Merkmal Temperatur 3℃ und die Aktivität fließen kennzeichnend für den Zustand der Wassertropfen auf der Fensterscheibe vor Ihnen. Wenn wir jedoch eines von dieser Welt mit Gewissheit sagen können dann, dass nichts bleibt wie es ist. Zustände ändern sich. Die Dinge gehen in andere Zustände über. Sie werden vielleicht bald ins Bett gehen und nach einer Weile schlafen. Ihr Zustand hat sich verändert von wach zu schlafend. Die Wassertropfen werden in der Nacht ebenfalls ihren Zustand ändern. Die Temperatur fällt unter Null und der Aggregatzustand der Wassertropfen geht von flüssig zu fest über. Diese Veränderungen der Zustände bezeichnen wir als Zustandsübergänge (engl. transition). Diese Übergänge geschehen unter bestimmten Bedingungen (engl. conditions). Sie werden nur in den Zustand schlafend übergehen, wenn Sie müde sind. Der Wassertropfen auf der Fensterscheibe wird nur dann zu Eis erstarren, wenn seine Temperatur den Gefrierpunkt erreicht hat. Die jetzt eingenommenen Zustände sind wiederum durch Attribute und Aktivitäten gekennzeichnet und werden unter bestimmten Bedingungen in andere Zustände übergehen. Das zustandsorientierte Durchdenken eines Systems ist für viele eine ungewohnte Übung. Doch viele Szenarien in eingebetteten Systemen aber auch in ganz normalen Anwendungssytemen sind dem Wesen nach Zustände und Zustandswechsel. Wir wollen dies im weiteren Zustandsautomat oder besser englisch state machine nennen. Die grafische Darstellung solcher Zustände und Zustandswechsel wird Zustandsdiagramm genannt.

Notationsübersicht Zustandsdiagramm (Auszug)

Notation	Bezeichnung	Beschreibung
	Startknoten	Ein Startknoten aktiviert Abläufe. Startknoten besitzen nur ausgehende Kanten.
Zustand entry / wird beim Betreten ausgeführt do / wird wiederholt während des Zusandes ausgeführt event / wird bei einem bestimmten Ereignis ausgeführt exit / wird beim Verlassen ausgeführt	Zustand	Zustände eines Objektes sind gekennzeichnet durch zustandsspezifische Aktivitäten beim Einnehmen, während und beim Verlassen des Zustandes (entry, do, exit)
	Zustandsübergang	Ein Zustandsübergang (transition) repräsentiert mindestens zwei Aktivitäten. Eine Aktivität vom Typ „exit“ beim alten und eine Aktivität vom Typ „entry“ beim neuen Zustand.
	Endknoten	Ein Endknoten beendet Abläufe und besitzt nur eingehende Kanten.

Die oben beschriebenen Zustände und Zustandsübergänge sind in der folgenden Darstellung mit den Mitteln des UML Zustandsdiagrammes abgebildet.

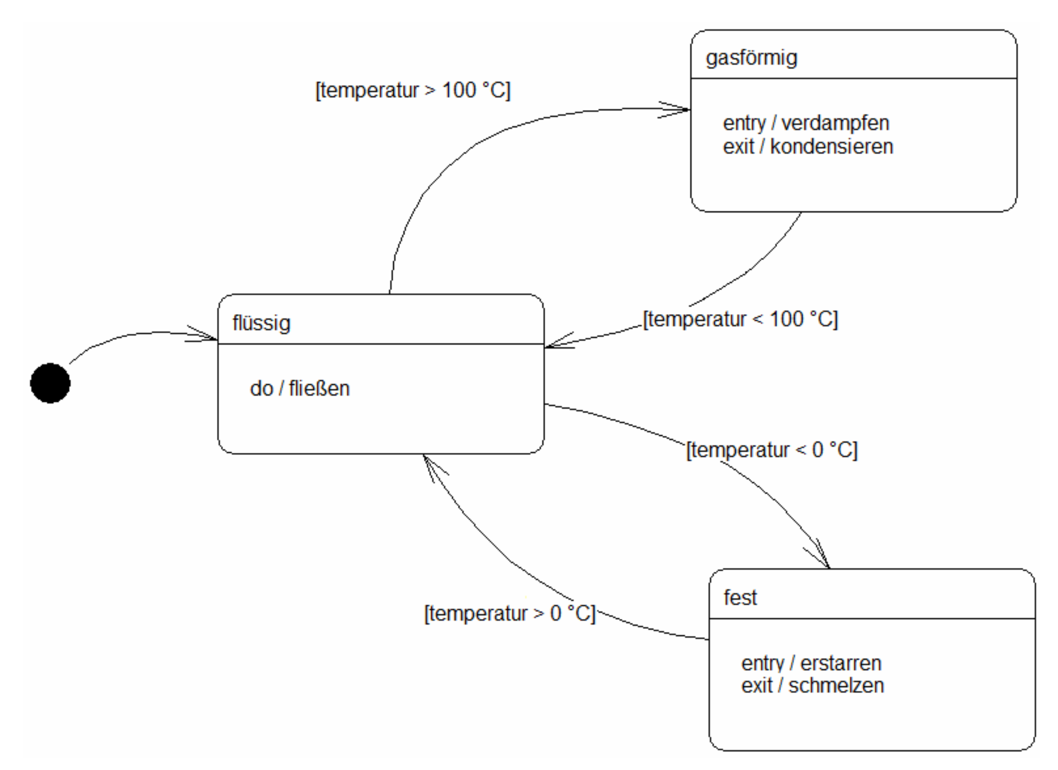
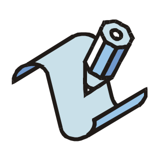


Abbildung: Zustände und Zustandsübergänge des Wassers bei Normaldruck

Zustandsorientierte Problemstellungen finden sich bei eingebetteten System recht häufig. Deshalb bietet sich auch an, diese Aufgabenstellungen mit dem Zustandsdiagramm nicht nur zu dokumentieren, sondern das Zustandsdiagramm als Realisierungsvorschrift zu verstehen. SiSy bietet zur Modellierung von Zustandsdiagrammen und zur Codegenerierung aus diesen ein spezielles Attribut im Klassendiagramm an. Zustandsdiagramme beschreiben mögliche Zustände und Zustandsübergänge eines Objektes. Es wird nur definiert was „innerhalb“ einer Instanz passiert. Interaktionen mit anderen Instanzen zeigt das Zustandsdiagramm nicht. Damit kann ein Zustandsmodell eindeutig einer Klasse zugeordnet werden. Das geschieht über das spezielle Zustandsattribut. Dieses ist auch gleichzeitig die Variable, in der sich eine Instanz während ihrer Lebenszeit den aktuellen Zustand speichert. Klassen können über mehr als ein Zustandsmodell verfügen.

Übung mit dem UML Zustandsdiagramm



Aufgabe:
Gehen Sie zurück in das Vorgehensmodell. Legen Sie ein neues Klassendiagramm mit dem Namen „Zustandstest“ an. Öffnen Sie das Diagramm und laden Sie die Diagrammvorlage „Grundgerüst“.

Hinweis:
Beachten Sie die korrekten Einstellungen (Hardware und Sprache).
Überprüfen Sie die Verbindungen auf dem Controllerboard. Es wird der Lichtsensor an ADC1 (Port F Bit 1) und die rote LED an Port D 0 erwartet.

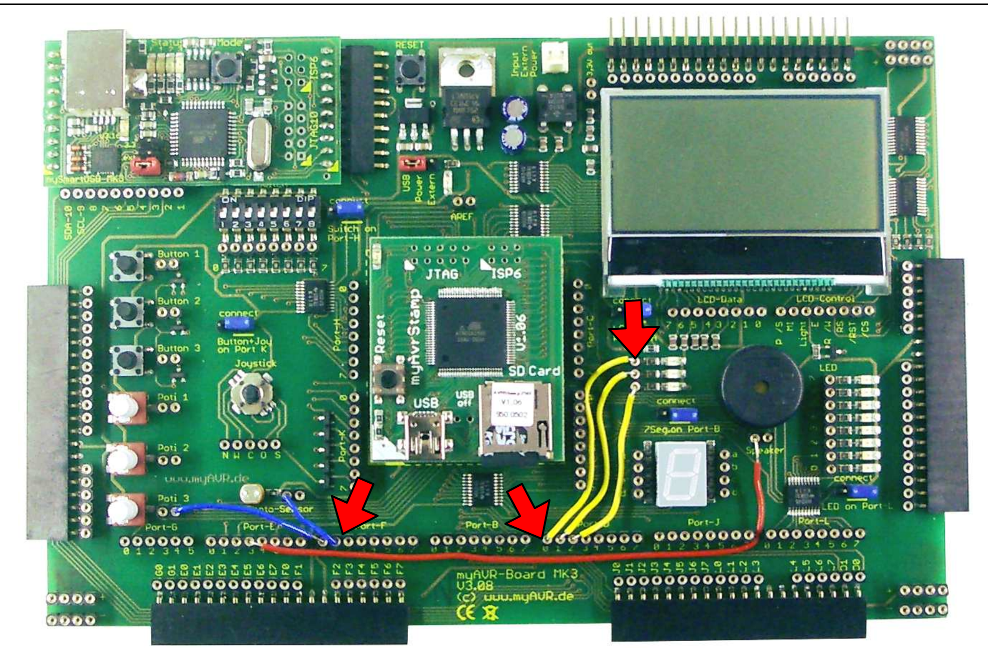


Abbildung: Verbindungen für die Übung Zustandstest

Die Aufgabe soll darin bestehen, den analogen Lichtsensor zu überwachen und entsprechend des Helligkeitswertes eine LED einzuschalten. Die Überwachung könnte durchaus die Klasse *Controller* übernehmen. Zur Demonstration kapseln wir jedoch die Überwachungsaufgabe in einem Wächter. Die Klasse *Controller* übernimmt lediglich die Aufgabe die Nachrichten zu verteilen.
Legen Sie ein Klasse mit dem Namen „Waechter“ an und aggrigieren Sie diese unter dem Namen „waechter“ in der Klasse *Controller*.

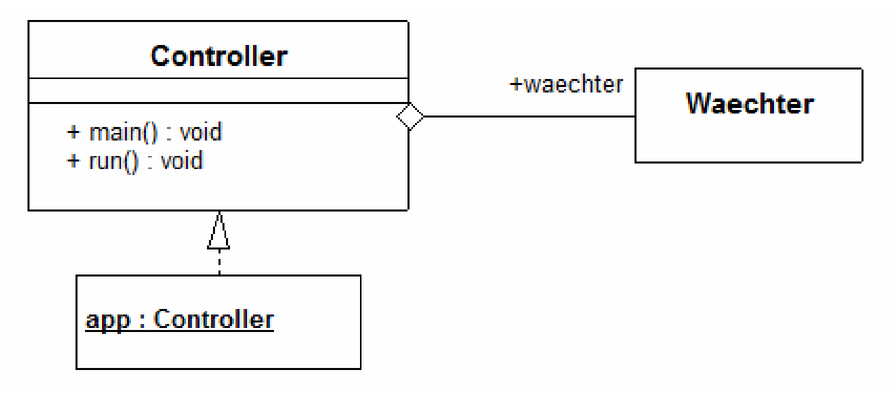


Abbildung: Anlegen der Klasse Wächter

Der Wächter ist verantwortlich für die Überwachung des Lichtsensors und das Schalten der LED. In den Paketen *myAVR_DigitalInOut* und *myAVR_ADC* finden wir geeignete Klassen, die eine einzelne digitale Ausgabe und das Einlesen eines Analogwertes realisieren. Ziehen Sie aus dem Navigator die genannten Pakete herein und die Klassen *Adc* sowie *DigitalOut*. Aggrigieren Sie die Klassen *Adc* und *DigitalOut* mit den Namen „lichtsensor“ und „led“ in der Klasse Wächter.

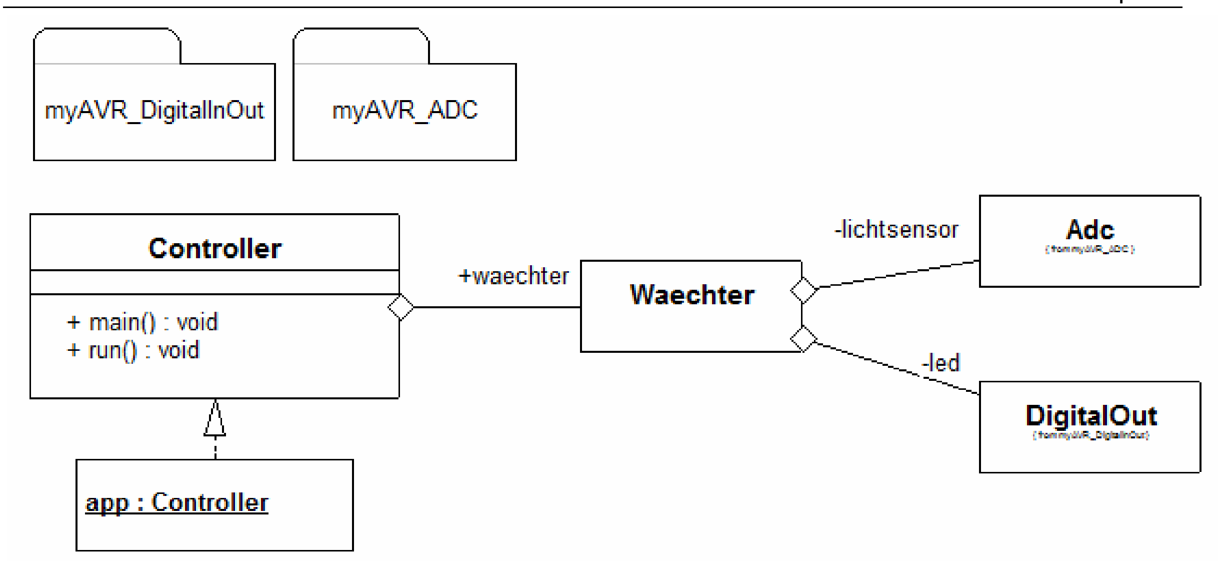


Abbildung: eingebundene Pakete und verwendete Bibliotheksklassen

Da der Wächter für Hardwarekomponenten verantwortlich ist benötigt er eine Operation „init“ von der aus es die Hardware konfigurieren kann. Des Weiteren muss der Wächter zyklisch die Möglichkeit bekommen seiner Überwachungsaufgabe nachzugehen. Das könnte man bei zeitkritischen Aufgaben per Timer-Interrupt lösen oder einfach aus der Hauptschleife heraus im Polling realisieren. Entscheidend ist jedoch, dass der Wächter dafür eine Operation zur Verfügung stellt. Diese Operation nennen wir „ueberwachen“. Ergänzen Sie die Klasse *Waechter* wie dargestellt. Hilfen zur Nutzung der Bibliotheksklassen finden Sie in den Beschreibungen der Klassen bzw. durch Studium der einsehbaren Klassendiagramme der benutzten Pakete. Vervollständigen Sie das Klassendiagramm wie gezeigt.

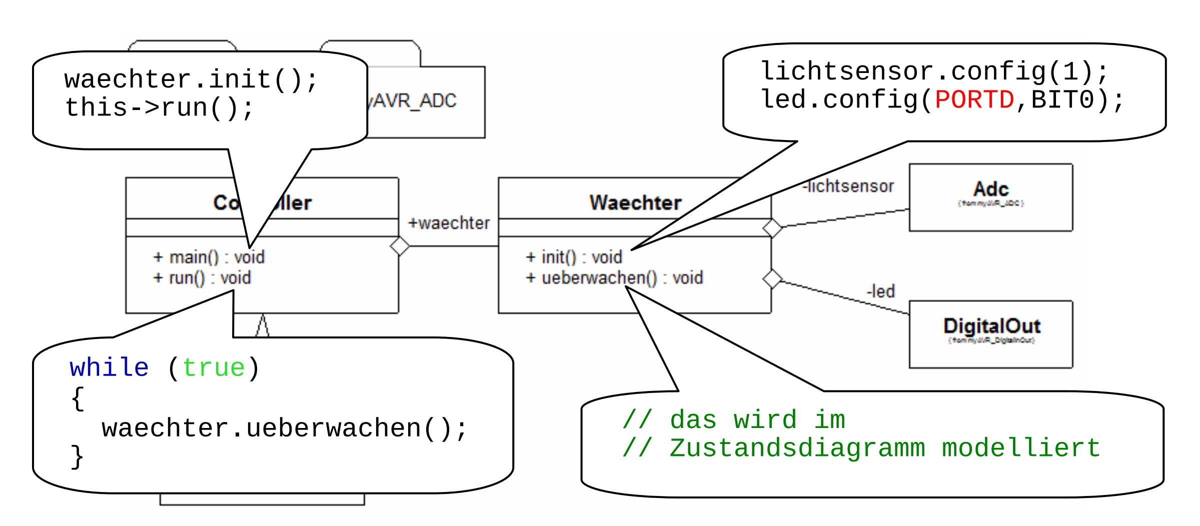


Abbildung: Wächterklasse mit den Operationen init und ueberwachen

Im nächsten Schritt wird der Klasse Wächter ein Element vom Typ „Zustandsattribut“ aus der Objektbibliothek eingefügt. Dieses Attribut wird im Weiteren mit einem Zustandsdiagramm hinterlegt. Als Namen für dieses Zustandsattribut legen Sie bitte „zustand“ fest.

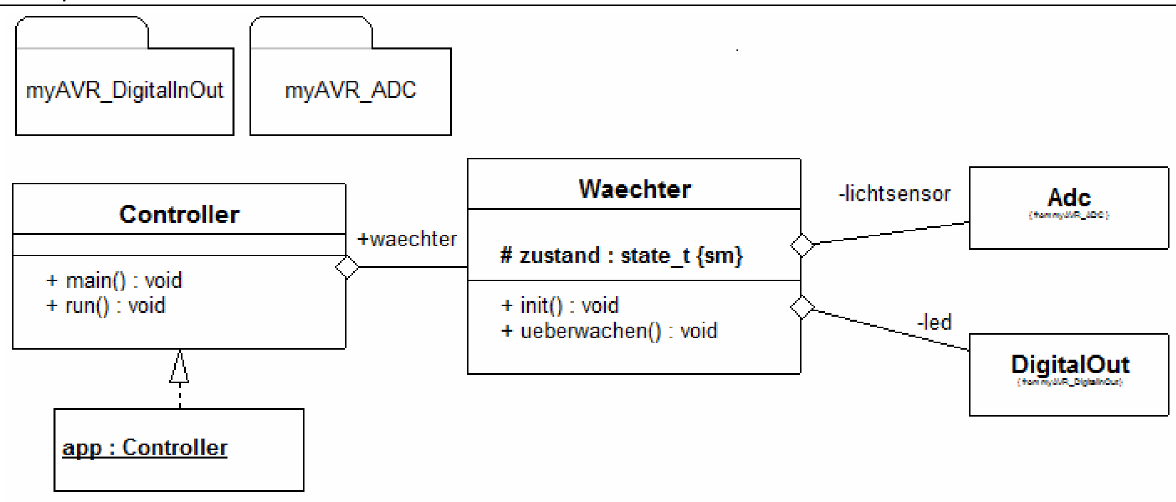


Abbildung: Klasse Waechter mit Zustandsattribut

Im Zustandsdiagramm werden nur diejenigen Operationen zugänglich gemacht, die eine Zuweisung zum Namensraum des Zustandsdiagramms (sm, state machine) erhalten haben. Das geschieht über eine Zusicherung in der Schreibweise WO::WAS. Die Operation ueberwachen soll die do-Aktivität im Zustandsdiagramm werden. Zur Erinnerung: Die do-Aktivität wird während eines Zustandes fortlaufend also zyklisch ausgeführt.

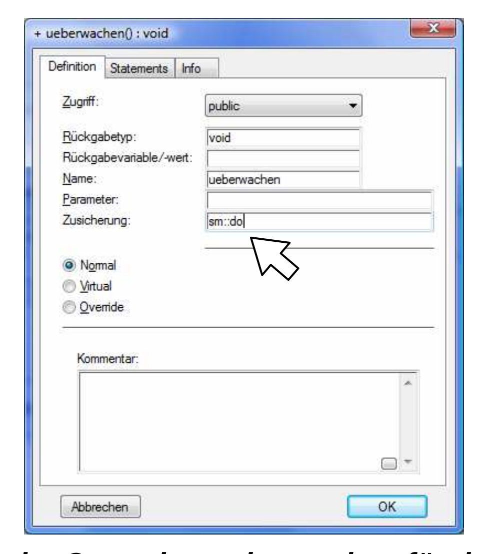


Abbildung: freigeben der Operation ueberwachen für das Zustandsdiagramm

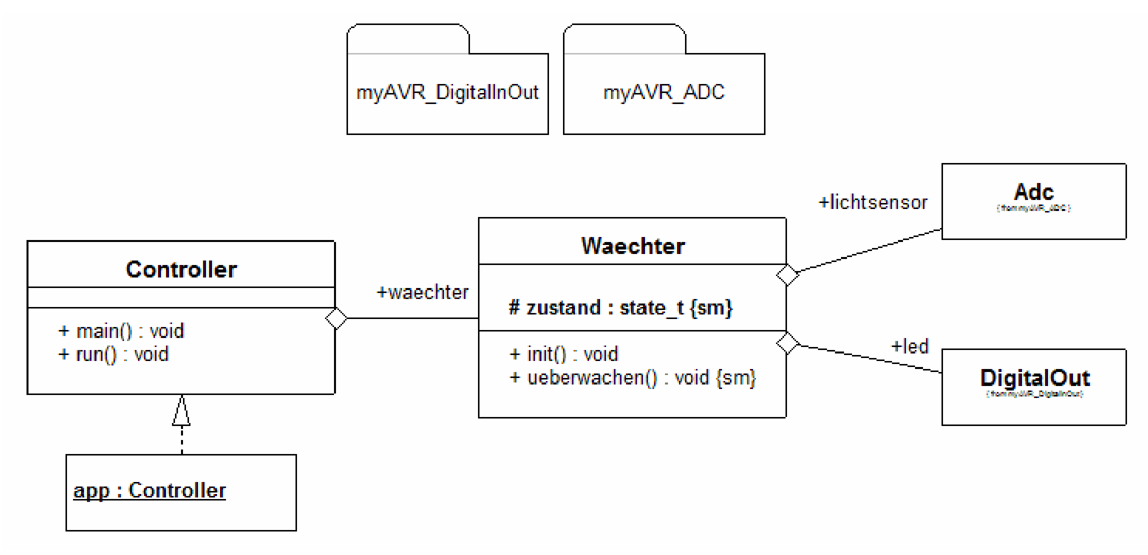


Abbildung: Klassendiagramm Zustandstest, beachten Sie die Zusicherungen {sm}

In das Zustandsdiagramm gelangen Sie über das Zustandsattribut. Selektieren Sie das Zustandsattribut und wählen im Kontextmenü (rechte Maustaste) „nach unten (öffnen)“. Es wird Ihnen eine Liste von Diagrammvorlagen für das Zustandsdiagramm angeboten. Wählen sie die Vorlage „UML Zustandsdiagramm mit 1-1“.

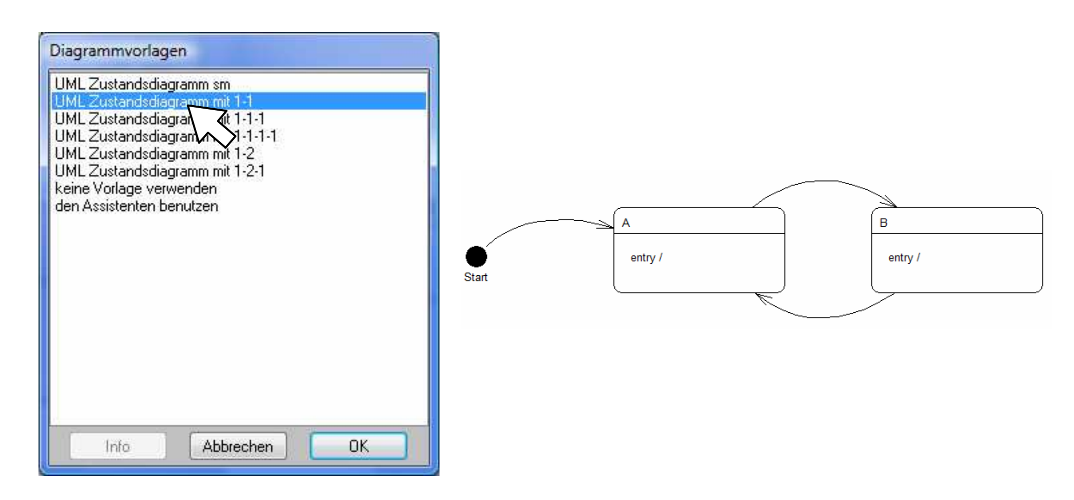


Abbildung: Vorlage für das Zustandsdiagramm wählen

Die gewählte Vorlage zeigt einen Startzustand (Startknoten) und zwei Zustände, die sich zyklisch abwechseln. Das Objekt nimmt nach dem Start immer einen der beiden Zustände ein. Unser Wächter soll nach dem Start zunächst in einen Zustand mit dem Namen „aus“ fallen. Danach können sich die Zustände „aus“ und „an“ je nach Umgebungslicht abwechseln. Die entry-Aktivitäten der Zustände sind für das Schalten der LED zuständig. Ändern Sie die Vorlage wie folgt:

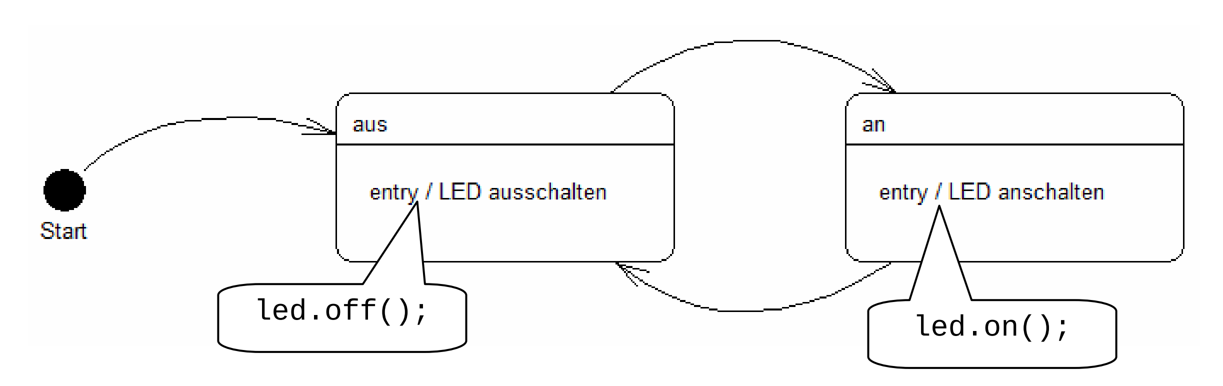


Abbildung: Vorlage bearbeiten

Die Konditionen für die Zustandswechsel werden auf den Kanten formuliert. Dabei muss angegeben werden, bei welcher Aktivität oder welchen Ereignis ein Zustandswechsel erfolgen kann. Des weiteren ist es möglich die Bedingung zu formulieren, wann der Zustandswechsel erfolgen darf und es kann ein beschreibender Text hinzugefügt werden.

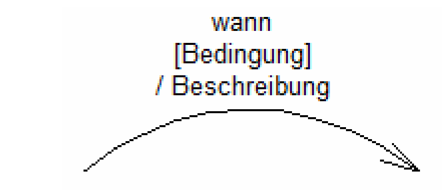


Abbildung: Notation eines Zustandswechsels

Der Wächter verfügt über einen Lichtsensor vom Typ *Adc*. Mit diesem kann der Helligkeitswert über die Operation *getAdc* ermittelt werden. Die Funktion liefert einen 10 Bit Wert. Als Grenzwert für das Umschalten legen wir 500 fest. Je nach Umgebungshelligkeit kann mit dieser Konstante der Arbeitspunkt und sogar eine Hysterese für die Umschaltlogik festgelegt werden. Selektieren Sie die Verbindungen und wählen im Kontextmenü (rechte Maustaste) „Definieren". Ergänzen Sie das Zustandsdiagramm wie gezeigt.

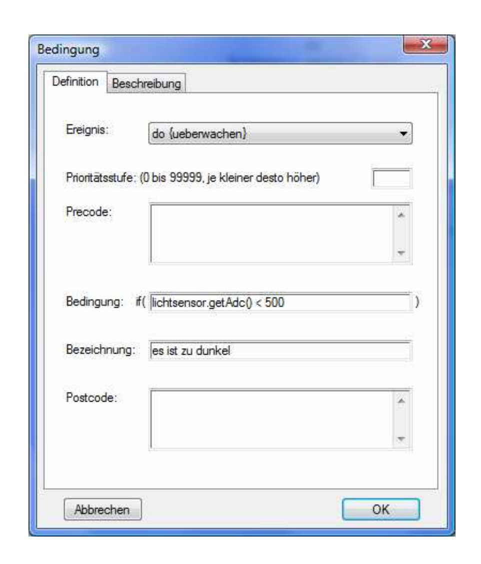


Abbildung: festlegen der Konditionen für den Zustandsübergang

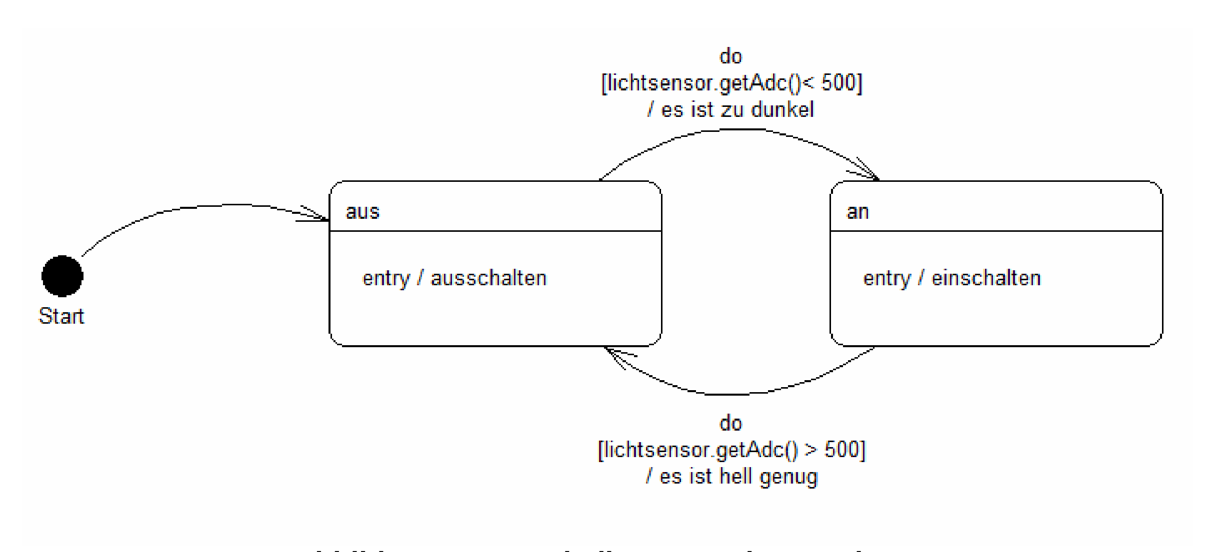


Abbildung: Zustandsdiagramm des Wächters

Gehen Sie zurück in das Klassendiagramm (ESC oder Kontextmenü „nach oben"). Erstellen, übertragen und testen Sie die Lösung. Beim Abdecken des Lichtsensors muss die rote LED aufleuchten. Vergleichen Sie den generierten Quellcode mit den Konstruktionszeichnungen.